Using Caché ObjectScript

Version 5.0.12
16 November 2004
# Table of Contents

## 1 Introduction to Caché ObjectScript
- 1.1 Features .................................................................................................................... 2
- 1.2 Language Overview ................................................................................................. 2
  - 1.2.1 Statements and Commands ............................................................................ 3
  - 1.2.2 Functions ........................................................................................................ 3
  - 1.2.3 Expressions ..................................................................................................... 4
  - 1.2.4 Variables ......................................................................................................... 4
  - 1.2.5 Operators ........................................................................................................ 4
- 1.3 Relationship with ANSI Standard M ....................................................................... 5

## 2 Syntax Rules
- 2.1 Case Sensitivity ........................................................................................................ 7
  - 2.1.1 Identifiers ........................................................................................................ 7
  - 2.1.2 Keyword Names ............................................................................................. 7
  - 2.1.3 Class Names ................................................................................................... 8
  - 2.1.4 Namespace Names ......................................................................................... 8
- 2.2 Invoking Commands and Functions ......................................................................... 8
- 2.3 White Space ............................................................................................................. 8
- 2.4 Comments ................................................................................................................ 9
  - 2.4.1 Single-Line Comments ................................................................................. 10
  - 2.4.2 Multiple-Line Comments ............................................................................. 10
  - 2.4.3 Retaining Comment Text in Object Routines ................................................. 10
- 2.5 Literals ................................................................................................................... 11
  - 2.5.1 String Literals ............................................................................................... 11
  - 2.5.2 Numeric Literals .......................................................................................... 11
- 2.6 Identifiers ............................................................................................................... 12
  - 2.6.1 Special Identifiers ......................................................................................... 13
  - 2.6.2 Dot (.) Character within Identifiers .............................................................. 13
- 2.7 Labels ..................................................................................................................... 13
- 2.8 Reserved Words ...................................................................................................... 14

## 3 Data Types and Values
- 3.1 Strings .................................................................................................................... 15
  - 3.1.1 Escaping Quotation Marks ........................................................................... 15
  - 3.1.2 Concatenating Strings ................................................................................. 15
14.2.4 Restore Journal Option of JOURNAL Utility .......................................... 181
14.3 System-Wide Issues with Transaction Processing ............................................. 182
  14.3.1 Backups with Transaction Processing ...................................................... 182
  14.3.2 Journaling with Transaction Processing ................................................... 182
  14.3.3 Globals to Avoid in Transactions .............................................................. 183
  14.3.4 Caché Network Transaction Processing ................................................... 183
  14.3.5 Asynchronous Error Notifications ............................................................ 183

15 Error Processing ........................................................................................................ 185
  15.1 How Error Processing Works ............................................................................. 185
    15.1.1 Internal Error-Trapping Behavior .......................................................... 186
    15.1.2 Current Context Level .......................................................................... 188
    15.1.3 $ETRAP Error Handlers ......................................................................... 189
    15.1.4 Error Codes ............................................................................................. 190
  15.2 Handling Errors with $ETRAP ........................................................................ 191
    15.2.1 Context-Specific $ETRAP Error Handlers .............................................. 191
    15.2.2 $ETRAP Flow of Control Options ........................................................... 193
  15.3 Handling Errors with $ZTRAP ........................................................................ 195
    15.3.1 Setting Up $ZTRAP ............................................................................... 195
    15.3.2 Using $ZTRAP ......................................................................................... 195
    15.3.3 Unstacking NEW Commands With Error Traps ...................................... 196
    15.3.4 $ZTRAP Flow of Control Options ........................................................... 196
  15.4 Handling Errors in an Error Handler ................................................................. 198
    15.4.1 Errors in a $ETRAP Error Handler .......................................................... 198
    15.4.2 Errors in a $ZTRAP Error Handler .......................................................... 199
    15.4.3 Error Information in the $ZERROR and $ECODE Special Variables .... 199
  15.5 Forcing an Error ............................................................................................... 200
    15.5.1 Setting $ECODE ...................................................................................... 200
    15.5.2 Creating Application-Specific Errors ....................................................... 200
  15.6 Processing Errors in Programmer Mode ........................................................... 200
    15.6.1 Understanding Error Message Formats ................................................... 201
    15.6.2 Understanding the Programmer Mode Prompt ....................................... 201
    15.6.3 Recovering from the Error ....................................................................... 201
    15.6.4 <EDITED> Errors .................................................................................... 202

16 Command-Line Routine Debugging ........................................................................ 205
  16.1 Debugging with the Caché Debugger ............................................................... 205
    16.1.1 Entering the Debugger from the Control Panel ....................................... 206
16.1.2 Using Breakpoints and Watchpoints ......................................................... 206
16.1.3 Establishing Breakpoints and Watchpoints .............................................. 207
16.1.4 Disabling Breakpoints and Watchpoints ................................................... 211
16.1.5 Delaying Execution of Breakpoints and Watchpoints .............................. 212
16.1.6 Deleting Breakpoints and Watchpoints .................................................... 212
16.1.7 Single-Step Breakpoint Actions ............................................................... 213
16.1.8 Tracing Execution ..................................................................................... 213
16.1.9 INTERRUPT Keypress and Break ........................................................... 216
16.1.10 Displaying Information About the Current Debug Environment ........... 216
16.1.11 Using the Debug Device ....................................................................... 219
16.1.12 Caché Debugger Example ................................................................. 220
16.1.13 Understanding Caché Debugger Errors .................................................. 221
16.2 Debugging With Break ................................................................................... 222
  16.2.1 Using Argumentless Break to Suspend Routine Execution ..................... 222
  16.2.2 Using Argumentless Break with a Condition .......................................... 223
  16.2.3 Using Argumented Break to Enable or Disable Interrupts ....................... 223
  16.2.4 Using Argumented Break to Suspend Routine Execution ........................ 223
  16.2.5 Enabling Single Stepping at the Previous Execution Level ..................... 225
  16.2.6 Understanding the Programmer Mode Prompt Information .................... 226
  16.2.7 Resuming Execution after a Break or an Error ........................................ 226
  16.2.8 The NEW Command in Programmer Mode ............................................ 228
  16.2.9 The QUIT Command in Programmer Mode ............................................ 229
  16.2.10 Caché Error Messages ........................................................................... 229
16.3 Using %STACK to Display the Stack ............................................................ 229
  16.3.1 Running %STACK ................................................................................... 230
  16.3.2 Displaying the Process Execution Stack .................................................. 230
  16.3.3 Understanding the Stack Display ............................................................. 231
16.4 Error Trap Utilities ....................................................................................... 235
  16.4.1 %ETN Application Error Trap ................................................................. 236
  16.4.2 %ERN Application Error Report ............................................................. 236
17 Controlling Caché from a Windows Client ...................................................... 239
  17.1 CctrlGetDirs ............................................................................................. 240
    17.1.1 Syntax ................................................................................................... 240
    17.1.2 Return Value ........................................................................................ 240
  17.2 CctrlConfigStatus ..................................................................................... 240
    17.2.1 Syntax ................................................................................................... 240
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.5.2 DTM Compatibility Mode Commands, Functions, and Special Variables</td>
<td>2</td>
</tr>
<tr>
<td>18.5.3 Database Conversion</td>
<td>7</td>
</tr>
<tr>
<td>18.6 MSM Language Compatibility</td>
<td>4</td>
</tr>
<tr>
<td>18.6.1 MSM Compatibility Mode Commands, Functions, and Special Variables</td>
<td>9</td>
</tr>
<tr>
<td>18.6.2 Database Conversion</td>
<td>278</td>
</tr>
</tbody>
</table>
List of Figures

Studio Syntax Checking ............................................................................................................... 9
Frames on a Call Stack ............................................................................................................. 187
$ETRAP Error Handlers .......................................................................................................... 193
$ZTRAP Error Handlers .......................................................................................................... 198
## List of Tables

<table>
<thead>
<tr>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date Formats</td>
<td>20</td>
</tr>
<tr>
<td>Caché ObjectScript Type Conversion Rules</td>
<td>24</td>
</tr>
<tr>
<td>ObjectScript Operators</td>
<td>28</td>
</tr>
<tr>
<td>Pattern Match Codes</td>
<td>51</td>
</tr>
<tr>
<td>Invoking Caché ObjectScript Functions</td>
<td>82</td>
</tr>
<tr>
<td>$ZF(-1) and ‘!’ Comparison</td>
<td>123</td>
</tr>
<tr>
<td>Caché Callin Interface Functions, Part I (High-Level Functions)</td>
<td>156</td>
</tr>
<tr>
<td>Caché Call-In Interface Functions, Part II (Low-Level Primitives)</td>
<td>158</td>
</tr>
<tr>
<td>Alternate Names for Functions and Data Types</td>
<td>162</td>
</tr>
<tr>
<td>Caché Transaction Commands</td>
<td>177</td>
</tr>
<tr>
<td>Error Prompts in Programmer Mode</td>
<td>226</td>
</tr>
<tr>
<td>%STACK Utility Information</td>
<td>232</td>
</tr>
<tr>
<td>Frame Types and Values Available</td>
<td>232</td>
</tr>
<tr>
<td>%ERN Options</td>
<td>237</td>
</tr>
<tr>
<td>Caché Language Modes</td>
<td>248</td>
</tr>
<tr>
<td>DSM-11 Compatibility Mode Open/Use Command Parameters</td>
<td>253</td>
</tr>
<tr>
<td>DSM-11 Compatibility Mode Set and Clear Status Bits</td>
<td>254</td>
</tr>
</tbody>
</table>
Caché ObjectScript is an object programming language designed for rapidly developing complex business applications. It is well-suited for a variety of applications including:

- Business logic
- Application integration
- Data processing

Caché ObjectScript source code is compiled into object code that executes within the Caché Virtual Machine. This object code is highly optimized for operations typically found within business applications, including string manipulations and database access. ObjectScript programs are completely portable across all platforms supported by Caché.

You can use Caché ObjectScript in any of the following contexts:

- Interactively from the command line of the Caché Terminal.
- As the implementation language for methods of Caché object classes.
- To create Caché ObjectScript routines: individual programs contained and executed within Caché.
- As the implementation language for Stored Procedures and Triggers within Caché SQL.
- As a server-side scripting language within a Caché Server Pages application.

Caché ObjectScript is completely compatible and interoperable with Caché's other native scripting language, Caché Basic.

To learn more about Caché ObjectScript, you can also refer to:
1.1 Features

Some of the key features of Caché ObjectScript include:

• Powerful built-in functions for working with strings.
• Native support for objects including methods, properties, and polymorphism.
• A wide variety of commands for directing control flow within an application.
• A set of commands for dealing with I/O devices.
• Support for multi-dimensional, sparse arrays: both local and global (persistent).
• Support for efficient, Embedded SQL.
• Support for indirection as well as runtime evaluation and execution of commands.

1.2 Language Overview

The following is an introduction to the major elements of Caché ObjectScript.

Caché ObjectScript does not define any reserved words: you are free to use any word as an identifier (such as a variable name). In order to accomplish this, Caché ObjectScript uses a set of built-in commands as well as special characters (such as “$” for function names) in order to distinguish identifiers from other language elements.

For example, to assign a value to a variable, you must use the **Set** command:

```
Set x = 100
Write x
```

Also note that white space can be significant within ObjectScript. In particular, a statement cannot start on the first character position on a line in order to distinguish it from a label:

```
MyLabel
Set x = 100
```
1.2.1 Statements and Commands

A Caché ObjectScript program consists of a number of statements. Each statement defines a specific action for a program to undertake. Each statement consists of a command and arguments.

Consider the following ObjectScript statement:

```
Write "Hello",!
```

The word **Write** is a command. It specifies the action to perform. The Write command does exactly what its name implies: it writes whatever you specify as its argument to the current device. In this case, **Write** writes the string “Hello” to the principal device (such as the terminal window if you are running a program within the Caché Terminal). Note that the “!” character is used as a short-hand for a carriage return for a text device.

There are a number of commands built-into Caché ObjectScript.

Every command within Caché ObjectScript has a long form and a short form (typically one or two characters). For example, the following code uses the short form of the **Write** command:

```
W "Hello"
```

The short form of a command name is simply a device for developers who do not like to type long commands. It is exactly equivalent to the long form. This document uses the long form of command names.

For more information on commands, refer to the **Commands** chapter or the individual reference page within the Caché ObjectScript Reference.

1.2.2 Functions

A **function** is a routine that performs a frequently-required operation (for example, converting a string to its equivalent ASCII code values). A function is invoked within a command line. This invocation passes parameters to the function, which uses these parameter values to perform some operation. The function then returns a single value that is the result of the operation. You can use a function any place you can use an expression. A function invoked upon an object is called a method. (Expressions and methods are described later in this chapter.)

In addition to its system-supplied functions, Caché ObjectScript allows you to write “procedures,” which are user-defined functions. The system-supplied functions are provided as part of Caché; they perform common string and data operations and each is described in the
Caché ObjectScript Reference. For information on defining and calling user-defined functions refer to User-Defined Code.

1.2.3 Expressions

An expression is any set of tokens that can be evaluated to yield a value. For example, the literal string, “hello”, is an expression. So is 1 + 2. Variables, such as x, or functions, such as $Length(x), are also expressions.

Within a program, you use expressions as arguments for commands and functions:

```
Set x = "Hello"
Write x,
Write 1 + 2,
Write $Length(x),
```

1.2.4 Variables

In ObjectScript, a variable is the name of a location in which a runtime value can be stored. Variables in Caché ObjectScript are untyped. Caché ObjectScript supports several kinds of variables:

- Local variables—A variable that is stored within the current Caché process and is deleted when a process ends.
- Globals—A persistent variable that is stored within the Caché database.
- Array variables—A variable with one or more subscripts. All non-system variables can be used as arrays, including local variables, globals, and object properties
- System variables—One of a special set of built-in variables that each specifies or indicates a particular aspect of the Caché operating environment.
- Object properties—A value associated with, and stored within, a specific instance of an object.

ObjectScript supports various operations on or among variables. These are described in the chapter on this topic.

1.2.5 Operators

Caché ObjectScripts defines a number of built-in operators such as addition ( “+” ) and multiplication ( “*” ). For details, refer to the Operators chapter.
1.3 Relationship with ANSI Standard M

Caché ObjectScript is a functional superset of the ANSI-standard M programming language. If you are an M programmer you can run your existing M applications on Caché with little or no change.

Caché ObjectScript offers a number of significant improvements over ANSI-standard M including:

- Integrated support for objects and object-oriented programming.
- Procedure and control blocks using { } syntax.
- Relaxed whitespace requirements.
- Many new functions.

You can take advantage of these new features in an evolutionary fashion, using them within your applications as you see fit.
This chapter describes the basic rules of ObjectScript syntax.

2.1 Case Sensitivity

Some parts of ObjectScript are case-sensitive while others are not. Generally speaking, the user-definable parts of ObjectScript are case-sensitive while the built-in command and function names are not.

2.1.1 Identifiers

User-defined identifiers (variable, routine, and label names) are case-sensitive. String, string, and STRING all refer to different variables. Global variable names are also case-sensitive, whether user-defined or system-supplied.

2.1.2 Keyword Names

Command, function, and system variable keywords (names) are case-insensitive. You can use Write, write, or WRITE; all refer to the same command.

Note: User-defined variables, routine names, and labels are case-sensitive. You must specify them exactly as defined.
2.1.3 Class Names

All identifiers related to classes (class names, property names, method names, etc.) are case-sensitive. For purposes of uniqueness, however, such names are considered to be case-insensitive; that is, two class names cannot differ by case alone.

2.1.4 Namespace Names

Namespace names are case-insensitive. Note however, that Caché always stores them in uppercase, and may return a namespace name to you in uppercase rather than the case which you specified.

2.2 Invoking Commands and Functions

ObjectScript syntax, in its simplest form, involves invoking commands on expressions, such as:

```
Write x
```

which invokes the Write command on the variable x (this displays the value of x). In the example above, x is an expression; an ObjectScript expression is one or more “tokens” that can be evaluated to yield a value. Each token can be a variable, the result of an operator's action (such as adding two numbers together), the return value that results from evaluating a function, some combination of these, and so on. The valid syntax for a statement involves its commands, functions, and operators and expressions; see each chapter for information on these.

2.3 White Space

Under certain circumstances, ObjectScript treats white space as syntactically meaningful. Unless otherwise specified, white space refers to blank spaces, tabs, and line feeds interchangeably. In brief, the rules are:

- White space must appear at the beginning of each line of code. The only exception to this is that a line can begin with a label (and not with white space); if a line has a label, there must be white space between the label and any code or comment.
• There must be one and only one space (not tab) between a command and its first argument; if a command uses a postconditional, there are no spaces between the command and its postconditional.

• If a postconditional expression includes any spaces, then the entire expression must be parenthesized.

• There can be any amount of white space between any pair of command arguments.

• If a line contains code and then a comment, there must be white space between them.

• Typically, each command appears on its own line, though you can enter multiple commands on the same line. In this case, there must be white space between them; if the first command is argumentless, then the second command must be preceded by two spaces or tabs (or one of each).

The Caché Studio provides built-in syntax checking, so that it will mark any illegal use of white space, such as the following insertion of multiple spaces before a command's first argument:

![Studio Syntax Checking](image)

### 2.4 Comments

It is good practice to use comments to provide in-line documentation in your routines. Comments are a valuable resource when modifying or maintaining code, especially if the modifications or maintenance is to be done by a person other than the original code writer.

There are two ways to specify a comment:

• A single line comment, indicated by a double slash (/) or a semicolon (;).
• A multiple line comment, beginning with a /* and ending with an */.

2.4.1 Single-Line Comments

You specify a single-line comment by a double slash (//), followed by the text of the comment. Caché ObjectScript interprets the text on that line that follows the double slash as a comment and does not attempt to execute it.

You can specify an entire line as a comment line, or append a comment to the end of label line or code line.

A *comment line* is a line that starts with white space followed by a double slash. You can insert comment lines anywhere within a routine.

If you append a comment to a code line, you must separate the identifying double slash from the last character of the command or command argument that precedes the comment by at least one whitespace character (space or tab).

Caché accepts a semi-colon (;) as a synonym for a double slash to indicate the beginning of a single-line comment.

2.4.2 Multiple-Line Comments

You specify a multiple-line comment by an opening slash-asterisk (/*), followed by the text of the comment for as many lines as needed, followed by a closing asterisk-slash (*.). Caché ObjectScript interprets the text between the /* and the */ as a comment and does not attempt to execute it.

You can begin a multiple-line comment on a separate line, or at the end of a label line or code line. A comment delimited by /* and */ may span multiple lines, or be embedded within a single line.

If you begin a comment in a code line, you must separate the identifying slash-asterisk (/*) from the last character in the command or command argument that precedes the comment by at least one whitespace character (space or tab).

2.4.3 Retaining Comment Text in Object Routines

When you compile a routine, the compiler strips out all comment text to produce a more compressed and efficient object version.

To retain comment text in an object code routine that Caché ObjectScript can recover (using the $Text function), even when the source routine is not loaded, use two semicolons (;;)
instead of a single semicolon. The two semicolons mean that $Text can retrieve the line much faster and retrieve it even though the source routine is not present.

### 2.5 Literals

A literal is a series of characters that represents a particular string or numeric value, such as “Hello” and “5” below:

```objectscript
Write "Hello"
Set x = 5
```

ObjectScript recognizes two types of literals:

- Numeric literals
- String literals

#### 2.5.1 String Literals

String literals are sets of zero or more of the 95 ASCII graphic characters (ASCII decimal values 32 to 126) enclosed in quotation marks (unlike string literals, numeric literals do not need quotation marks). String literals consist of numbers, uppercase and lowercase letters, or symbolic characters (such as $, #, %, or &) enclosed in quotation marks. The value of a string literal is constant and determined by the characters, including symbolic characters and spaces enclosed in quotation marks.

#### 2.5.2 Numeric Literals

Numeric literals are strings that ObjectScript evaluates as numbers. ObjectScript treats as a number any string that contains only the following:

- The digits 0 through 9
- The Unary Minus operator (-)
- The Unary Plus operator (+)
- The period or decimal point character
- The Letter “E” (used in exponential notation)

ObjectScript can work with the following types of numbers:
Integers (whole numbers such as 100)

Decimal numbers (real numbers such as 3.767)

Decimal fractions (real numbers such as .0442)

Numbers placed in exponential notation (such as 2.8E2)

To specify exponential notation in ObjectScript, use the following format:

\[-\]\text{mantissa}E[-]\text{exponent}

where

\begin{tabular}{|l|p{10cm}|}
\hline
\text{-} & The optional Unary Minus operator used with negative numbers. \\
\text{mantissa} & The decimal or integer number. \\
\text{E} & An operator delimiting the exponent (can be upper or lower case). \\
\text{-} & The optional Unary Minus operator used with a negative exponent. \\
\text{exponent} & The integer exponent (the power of 10). \\
\hline
\end{tabular}

For example, to represent the number 10, use $1E1$; and to represent number 280, use $2.8E2$.

### 2.6 Identifiers

An \textit{identifier} is the name of a variable, function, or label.

Legal identifiers have the following characteristics:

- The first character is a letter (upper or lower case) or a Unicode character greater than 127 (on a Unicode version of Caché).
- Subsequent characters can be any letter, digit, or a Unicode character greater than 127 (on a Unicode version of Caché).
- No punctuation of any kind is allowed within an identifier (with some exceptions noted below).

The following are legal identifiers:
2.6.1 Special Identifiers

Special identifier names are used for certain items. These special names start with a punctuation character:

- “^” —indicates the name of a global (a persistent, multidimensional array):

  ^array

- “%” —indicates a “% variable”. % variables have special scoping rules (they are always visible) and are typically used for special system variables or values that need to be visible on an application-wide basis. Routine names can also start with “%”.

2.6.2 Dot (.) Character within Identifiers

Certain identifiers can contain one or more “.” characters. These include:

- A global name (but not a local variable name) may include one or more “.” characters:

  “^” —indicates the name of a global (a persistent, multidimensional array):

  ^MyApp.PersonD

- A routine name may include one or more “.” characters.

The “.” must not appear at the beginning or end of the name.

2.7 Labels

Any line of ObjectScript code can optionally include a label (also known as a tag). A label is a string that appears at the beginning of a line and serves as a handle for referring to the line and subsequent code. Labels have the following rules:

- They must begin with an alphanumeric or the percent character; the second and subsequent characters must be alphanumerics.
- They can be up to 31 characters long.
• A line can contain only a label; if code or a comment follows the label, they must be preceded by a space or tab character.

• They are case-sensitive.

The following are all unique labels:

maximum
Max
MAX
86
agent86
86agent
%control

Labels are useful for managing flow control, since you can use the label to invoke the code that follows it. At the same time, a label does not intrinsically define an encapsulated unit of code; this means that, once the labelled code executes, control passes to the next labelled set of code, if there is one. For instance, in the following code:

```
label22
  Set x = 22
  Write !,"x = ",x
label23
  Set x = 23
  Write !,"x = ",x
```

control passes from the code under “label22” to that under “label23,” which results in the value of x being changed from 22 to 23. To avoid this situation, exit from the code where appropriate, so that the code is:

```
label22
  Set x = 22
  Write !,"x = ",x
  Quit
label23
  Set x = 23
  Write !,"x = ",x
  Quit
```

2.8 Reserved Words

There are no reserved words in Caché ObjectScript; you can use any valid identifier as a variable name, function name, or label. At the same time, it is best to avoid using identifiers that are command names, function names, or other such strings. Also, since ObjectScript code includes support for embedded SQL, it is prudent to avoid naming any function, object, variable, or other entity with an SQL reserved word, as this may cause difficulties elsewhere.
Data Types and Values

Caché ObjectScript is a typeless language—you do not have to declare the types of variables. Any variable can have a string, numeric, or object value.

3.1 Strings

A string is a set of characters: letters, digits, punctuation, etc. You can define a string literal by placing text within a matched set of quotation marks ("):

Set string = "This is a string."
Write string

3.1.1 Escaping Quotation Marks

You can include a " (double quote) character within a string literal by preceding it with a " character:

Set string = "This string has "quotes" in it."
Write string

There are no other escape sequences within ObjectScript string literals.

3.1.2 Concatenating Strings

You can concatenate two strings into a single string using the _ concatenation operator:

Set string = "Hello" _ " Goodbye"
Write string
3.2 Numbers

Numeric literal values have the following characteristics:

- They can contain the decimal numbers 0-9, a decimal point (.), any number of leading plus and minus signs, and the letter “E” or “e” (to delimit an exponent).
- They can contain leading or trailing zeros, which are stripped off from the results.
- They cannot contain commas, currency symbols, or other characters. They cannot contain blank spaces, except before or after operators (such as plus and minus signs). Each operand cannot contain more than one decimal point or more than one exponentiation character (“E” or “e”).
- An exponentiation character (“E” or “e”) must be immediately followed by either a whole number, or a single plus or minus sign followed by a whole number. It cannot be followed by a blank space, a decimal point, multiple plus and minus signs, or a fractional number. An exponent can contain leading or trailing zeros.
- Numbers of greater than 19 digits, or exponents greater than 130 may give unpredictable results.

3.3 Objects

An object value refers to a instance of an in-memory object. You can assign an object value to any local variable:

```ObjectScript
Set person = Sample.Person
Write person
```

You can refer to the methods and properties of an object using dot syntax:

```ObjectScript
Set person.Name = "El Vez"
```

You can determine if a variable contains an object using the IsObject function:
Set str = "A string"
Set person = ##class(Sample.Person).%New()

If $IsObject(person) {
    Write "Person is an object."
} Else {
    Write "Person is not an object."
}

If $IsObject(str) {
    Write "String is an object."
} Else {
    Write "String is not an object."
}

You cannot use assign an object value to a global. Doing so results in a runtime error.

Assigning an object value to a variable (or object property) has the side effect of incrementing the object's internal reference count. When the number of references to an object reaches 0, Caché automatically destroys the object (invoke its %OnClose method and remove it from memory).

## 3.4 Persistent Multi-dimensional Arrays (Globals)

A global is a sparse, multidimensional database array. A global is not different from any other type of array, with the exception that the global variable name starts with a caret (^). Data can be stored in a global with any number of subscripts; subscripts in Caché are typeless.

The following is an example of using a global. Once you set the global ^x, you can examine its value:

Set ^x = 10
Write "The value of ^x is: ", ^x

For more information on globals, see the “Multidimensional Arrays” chapter in this document and the Using Caché Multi-Dimensional Storage document.

## 3.5 Undefined Values

ObjectScript variables do not need to be declared or defined. By simply assigning a value to a variable it is defined, but, until this first assignment, all references to this variable are
undefined. You can use the $Data or $Get functions to determine if a variable is defined or undefined.

### $Data

$Data takes one or two arguments. With one argument, it simply tests if a variable has a value:

```objectscript
Write "Does ""MyVar"" exist?",!
if $Data(MyVar) {
  Write "It sure does!"
} Else {
  Write "It sure doesn't!"
}
Set MyVar = 10
Write !,"How about now?",!
if $Data(MyVar) {
  Write "It sure does!"
} Else {
  Write "It sure doesn't!"
}
```

$Data returns a boolean that is True (1) if the variable has a value (that is, contains data) and that is False (0) if the variable has no value (that is, contains no data). With two arguments, it performs the test and sets the second argument's variable equal to the tested variable's value:

```objectscript
If $Data(Var1,Var2) {
  Write "Var1 has a value of ",Var2,"."!
} Else {
  Write "Var1 is undefined."!
}
Set Var1 = 3
If $Data(Var1,Var2) {
  Write "Var1 has a value of ",Var2,"."!
} Else {
  Write "Var1 is undefined."!
}
```

Instead of using $Data, you can also test if a variable is defined or undefined using the $Get function.

```objectscript
If $Get(MyVar) {
  Write "MyVar has a value of ",MyVar,"."!
} Else {
  Write "MyVar is undefined."!
}
Write "MyVar has a value of ",$Get(MyVar,3),"."!
```

In the If construct, $Get with one argument returns 0 (False) if MyVar has no value. With two arguments, the second argument of $Get provides a default value to return if its argument is undefined; note that it does not set the value of the first argument's variable.
3.6 Boolean Values

In certain cases, such as when used with logical commands or operators, a value may be interpreted as a boolean (true or false) value. In such cases, a value is interpreted as true if it evaluates to a non-zero numeric value or false if it evaluates to a zero numeric value.

For example, the following values are interpreted as true:

```
1
10
"1 banana"
1 + 1
-1
```

The following values are interpreted as false:

```
0
"
"one banana"
1 - 1
```

3.7 Dates

ObjectScript has no built-in date type; instead it includes a number of functions for operating on and formatting date values represented as strings. These date formats include:
**Date Formats**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Horolog</td>
<td>This is the format returned by the $Horolog ($H) special variable. It is a string containing two, comma separated integers: the first is the number of days since December 31, 1840; the second is the number of seconds since midnight of the current day. There are a number of functions for formating and validating dates in this format.</td>
</tr>
<tr>
<td>ODBC Date</td>
<td>This is the format used by ODBC and many other external representations. It is a string of the form: “YYYY-MM-DD HH:MM:SS”. ODBC date values will collate; that is if you sort data by ODBC date format, it will automatically be sorted in chronological order.</td>
</tr>
<tr>
<td>System Time</td>
<td>This is the format returned by the $ZHorolog ($ZH) special variable. It is a floating point number containing the number of seconds (and parts thereof) that the system has been running. Typically this format is used for timing and testing operations.</td>
</tr>
</tbody>
</table>

The following example shows how you can use the different date formats:

```ObjectScript
Set now = $Horolog
Write "Current time and date ($H): ",now,!

Set odbc = $ZDateTime(now,3)
Write "Current time and date (ODBC): ",odbc,!

Set time = $ZH
Write "Current system time ($ZH): ",time,!
```
4

Variables

A variable is the name of a location in which a value can be stored. Within Caché ObjectScript, a variable does not have type associated with it and you do not have to declare it.

4.1 Categories of Variables

Within Caché ObjectScript, there are several categories (kinds) of variables. Each of these is used for a specific purpose and may have different scoping rules. Within a program you can use any category of variable in the same way (such as assigning values or passing them as function arguments).

4.1.1 Local Variables

A local variable is a variable that is stored within the current Caché process. When a process ends, all of its local variables are deleted.

There are three kinds of local variables:

- **Private variables.** Any variable used within a procedure block is automatically a private variable and is only visible within that procedure block. By default, all object methods created with Caché Studio use procedure blocks (the ProcedureBlock class keyword is set within the class definition) and so, by default, all variables created in methods are private variables.

  You cannot use the New command on a private variable.
• **Public variables.** A public variable, once defined, is visible to any code within the current process unless one of the following is true: (1) the variable has been redefined using the New command, or (2) the program has entered a procedure block.

• **% Variables.** A variable whose name starts with “%” is always treated as a public variable. This makes it possible to define special variables that are visible to all code within a process.

Refer to the section on Variable Scoping for more information.

### 4.1.2 Globals

A global is a special kind of variable that is automatically stored within the Caché database.

Within a Caché ObjectScript program, you can use a global in the same way as any other variable. Syntactically, a global name is distinguished by prepending a “^” character:

```objectscript
Set local = "This is a local variable."
Set "global = "This will be stored in the database."
```

For more information on globals, refer to Using Multi-Dimensional Storage.

### 4.1.3 Array Variables

An array variable is simply a variable with one or more subscripts. Any variable (with the exception of built-in system variables) can be used as an array including local variables, globals, and object properties:

```objectscript
Set a(1) = "An array"
Set a(1,1,1) = "Another array"
Set ^a(1) = "A global array"
Set obj.a(1) = "A multi-dimensional array property"
```

### 4.1.4 System Variables

Caché ObjectScript includes a number of built-in system variables (also referred to as special variables) that are used to make certain system information available to applications. The value of a special variable is set to the current state of some aspect of your operating environment. All special variable start with a “$” character. For example, consider the following code:

```objectscript
Set time = $HOROLOG
Write time,!
```
This command references the special variable $HOROLOG. $HOROLOG stores the current system date and time. The Set command uses this special variable to set the user-defined variable time to this value.

Other examples of system variables include:

Write "$JOB = ", $JOB,!  // Current process ID
Write "$ZVERSION = ", $ZVERSION,! // Version info

Most system variables are read-only, though some, such as $IO are read-write. Refer to the Caché ObjectScript Reference for a list and detailed descriptions of the system variables.

### 4.1.5 Object Properties

An object property is a value associated with, and stored within, a specific instance of an object. Strictly speaking, an object property is not a variable, but syntactically you can use an object property in exactly the same way as any other variable:

```objectscript
// Create an Address object
Set address = ##class(Sample.Address).%New()

// Use the properties of the object
Set address.City = "Boston"
Write "City: ", address.City,
```

### 4.2 Variable Typing and Conversion

Variables in Caché ObjectScript are, as in JavaScript, VBScript, and Caché Basic, untyped. This means that you can assign a string value to a variable and, later on, a numeric value to the same variable. Internally, Caché does treat strings, integers, numbers, and object as different types but this is not visible to the application programmer. Caché automatically converts the value of variables to other types, based on the context in which it is used.

Some examples:
// set some variables  
Set a = "This is a string"  
Set b = "3 little pigs"  
Set int = 22  
Set num = 2.2  
Set obj = ##class(Sample.Person).%New()  

// Display them  
Write "$a: "$a,!  
Write "$b: "$b,!  
Write "$int: "$int,!  
Write "$num: "$num,!  
Write "$obj: "$obj,!  

// Now use them as other "types"  
Write "$+a: "$+a,!  
Write "$+b: "$+b,!  
Write "$int _ " abc": "$int _ " abc",!  
Write "$num _ " abc": "$num _ " abc",!  
Write "$+obj: "$+obj,!

Caché converts types as follows:

**Caché ObjectScript Type Conversion Rules**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Object</td>
<td>Not allowed.</td>
</tr>
<tr>
<td>Number</td>
<td>String</td>
<td>The number is converted to a string of numeric characters that corresponds to the number's value: \texttt{22} becomes \texttt{&quot;22&quot;}.</td>
</tr>
<tr>
<td>Object</td>
<td>Number</td>
<td>The object reference is converted to an integer whose value is the object instance number.</td>
</tr>
<tr>
<td>Object</td>
<td>String</td>
<td>The object reference is converted to a string in the form: \texttt{&quot;oref@classname&quot;} where \texttt{oref} is the object instance number and \texttt{classname} is the type of object.</td>
</tr>
<tr>
<td>String</td>
<td>Number</td>
<td>The string is parsed left-to-right until a non-numeric character is encountered. The leading numeric characters are converted to the corresponding numeric value. If there are no leading numeric characters, the string is converted to 0. For example, \texttt{&quot;123&quot;} converts to \texttt{123}, \texttt{&quot;123abc&quot;} converts to \texttt{123}, and \texttt{&quot;abc123&quot;} converts to 0.</td>
</tr>
<tr>
<td>String</td>
<td>Object</td>
<td>Not allowed.</td>
</tr>
</tbody>
</table>

### 4.2.1 Object Values

An object value refers to a instance of an in-memory object. You can assign an object value to any local variable:
Note: The value of person is that of an object reference (OREF) converted into a string. This string or its value cannot be used to load an object from the database.

You can refer to the methods and properties of an object using dot syntax:

Set person.Name = "El Vez"

You can determine if a variable contains an object using the $IsObject function:

Set str = "A string"
Set person = ##class(Sample.Person).%New()
Write "Is string an object? ", $IsObject(str),!
Write "Is person an object? ", $IsObject(person),!

You cannot use assign an object value to a global. Doing so will result in a runtime error.

Assigning an object value to a variable (or object property) has the side effect of incrementing the object's internal reference count. When the number of references to an object reaches 0, Caché will automatically destroy the object (invoke its %OnClose method and remove it from memory). For example:

Set person = ##class(Sample.Person).%New() // one reference to Person
Set alias = person // two references
Set person = "" // 1 reference
Set alias = "" // no references left, object destroyed

4.3 Variable Declaration and Scope

Unlike other languages, you do not declare variables in Caché ObjectScript.

Variable scoping determines when a variable is “visible” to a program. In Caché ObjectScript, there are two sets of rules for variable scoping:

- The standard (and newer) scoping mechanism, which is based on procedure blocks. This is the preferred mechanism for new applications and is the default used by the Caché Studio. Within a procedure block, all non-% variables are private variables
- An older, legacy scoping mechanism. This is provided for compatibility with older versions. Outside of a procedure block, all variables are public variables.

These mechanisms are described greater detail in the User-defined Code chapter.
5

Operators and Expressions

Caché supports many different operators, which perform various actions, including mathematical actions, logical comparisons, and so on. Operators act on expressions, which are variables or other entities that ultimately evaluated to a value. This chapter describes expressions and the various ObjectScript operators. It contains the following topics:

- Introduction to Operators and Expressions
- Arithmetic Operators
- Logical Comparison Operators
- String Operators
- Numeric Relational Operators
- String Relational Operators
- Pattern Matching
- Indirection

5.1 Introduction to Operators and Expressions

Operators are symbolic characters that specify the action to be performed on their associated operands. Each operand consists of one or more expressions or expression atoms. When used together, an operator and its associated operands have the following form:

[operand] operator operand
Some operators take only one operand and are known as unary operators; others take two operands and are known as binary operators.

An operator and any of its operands taken together constitute an expression. Such expressions produce a result that is the effect of the operator on the operand(s). They are classified based on the types of operators they contain.

- An *arithmetic expression* contains arithmetic operators, gives a numeric interpretation to the operands, and produces a numeric result.
- A *string expression* contains string operators, gives a string interpretation to the operands, and produces a string result.
- A *logical expression* contains relational and logical operators, gives a logical interpretation to the operands, and produces a boolean result: TRUE (1) or FALSE (0).

ObjectScript includes the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Object property or method access.</td>
</tr>
<tr>
<td>()</td>
<td>Array index or function call arguments.</td>
</tr>
<tr>
<td>+</td>
<td>Addition (Binary), Positive (Unary)</td>
</tr>
<tr>
<td>–</td>
<td>Subtraction (Binary), Negative (Unary)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>\</td>
<td>Integer division</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>#</td>
<td>Modulus (remainder)</td>
</tr>
<tr>
<td>_</td>
<td>Concatenation</td>
</tr>
<tr>
<td>'</td>
<td>Logical complement (NOT)</td>
</tr>
<tr>
<td>=</td>
<td>Test for equality, Assignment</td>
</tr>
<tr>
<td>'≡'</td>
<td>Test for non-equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>'＞'</td>
<td>Not greater than (less than or equal)</td>
</tr>
<tr>
<td>Operator</td>
<td>Operation Performed</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>Not less than (greater than or equal)</td>
</tr>
<tr>
<td>[</td>
<td>Contains</td>
</tr>
<tr>
<td>]</td>
<td>Follows</td>
</tr>
<tr>
<td>]]</td>
<td>Sorts After</td>
</tr>
<tr>
<td>&amp;&amp;, &amp;</td>
<td>Logical AND (&amp;&amp; is “short-circuit” AND)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>Indirection</td>
</tr>
<tr>
<td>?</td>
<td>Pattern Match</td>
</tr>
</tbody>
</table>

These are described in more detail in the following sections.

## 5.1.1 Operator Precedence

Operator precedence in ObjectScript is strictly left-to-right; within an expression operations are performed in the order in which they appear. This is different from other languages in which certain operators have higher precedence than others. You can use explicit parentheses within an expression to force certain operations to be carried ahead of others.

```
WRITE "1 + 2 * 3 = ", 1 + 2 * 3, ! // returns 9
WRITE "2 * 3 + 1 = ", 2 * 3 + 1, ! // returns 7
WRITE "1 + (2 * 3) = ", 1 + (2 * 3), ! // returns 7
WRITE "2 * (3 + 1) = ", 2 * (3 + 1), ! // returns 8
```

### 5.1.1.1 Parentheses and Precedence

You can change the order of evaluation by nesting expressions within each other with matching parentheses. The parentheses group the enclosed expressions (both arithmetic and relational) and control the order in which Caché ObjectScript performs operations on the expressions. Consider the following expression:

```
Set TorF = ((4 + 7) > (6 + 6)) // False
Write TorF
```

Here, because of the parentheses, four and seven are added, as are six and six; this results in the logical expression 11 > 12, which is false. Compare this to:

```
Set Value = (4 + 7 > 6 + 6) // 7
Write Value
```
In this case, precedence proceeds from left to right, so four and seven are added. Their sum, eleven, is compared to six; since eleven is greater than six, the result of this logical operation is one (TRUE). One is then added to six, and the result is seven.

Note that the precedence even determines the result type, since the first expression's final operation results in a boolean and the second expression's final operation results in a numeric.

The following example shows multiple levels of nesting:

```plaintext
WRITE 1+2*3-4*5,! // returns 25
WRITE 1+(2*3)-4*5,! // returns 15
WRITE 1+(2*(3-4))*5,! // returns -5
WRITE 1+(((2*3)-4)*5),! // returns 11
```

Precedence from the innermost nested expression and proceeds out level by level, evaluating left to right at each level.

### 5.1.1.2 Functions and Precedence

Some types of expressions, such as functions, can have side effects. Suppose you have the following logical expression:

```plaintext
IF var1 = ($$ONE + (var2 * 5)) {
    DO ^Test
}
```

Caché ObjectScript first evaluates `var1`, then the function `$ONE`, then `var2`. It then multiplies `var2` by 5. Finally, Caché ObjectScript tests to see if the result of the addition is equal to the value in `var1`. If it is, it executes the `DO` command to call the `Test` routine.

As another example, consider the following logical expression:

```plaintext
SET var8=25, var7=23
IF var8 = 25 * (var7 < 24) {
    WRITE !,"True"
} ELSE {
    WRITE !,"False"
}
```

This expression compares `var7` for a value less than 24 and then multiplies 25 by the result (1 for TRUE, 0 for FALSE). Finally, it tests to see whether the result (25 or 0) is equal to the value in `var8`.

### 5.1.2 Expressions

An ObjectScript expression is one or more “tokens” that can be evaluated to yield a value. The simplest expression is simply a literal or variable:
Set expr = 22
Set expr = "hello"
Set expr = x

You can create more complex expressions using arrays, operators, or one of the many ObjectScript functions:

Set expr = +x
Set expr = x + 22
Set expr = array(1)
Set expr = "data(\"x\", 1"
Set expr = $Length(x)

An expression may consist of, or include, an object property, instance method call, or class method call:

Set expr = person.Name
Set expr = obj.Add(1,2)
Set expr = ##class(MyApp.MyClass).Method()

You can directly invoke an ObjectScript routine call within an expression by placing $$ in front of the routine call:

Set expr = $$MyFunc^MyRoutine(1)

Expressions can be classified according to what kind of value they return:

• An **arithmetic expression** contains arithmetic operators, gives a numeric interpretation to the operands, and produces a numeric result:

  Set expr = 1 + 2
  Set expr = +x
  Set expr = a + b

  Note that a string used within an arithmetic expression is evaluated as a numeric value (or 0 if it is not a valid numeric value). Also note that using the unary addition operator (+) will implicitly convert a string value to a numeric value.

• A **string expression** contains string operators, gives a string interpretation to the operands, and produces a string result.

  Set expr = "hello"
  Set expr = "hello" _ x

• A **logical expression** contains relational and logical operators, gives a logical interpretation to the operands, and produces a boolean result: TRUE (1) or FALSE (0):

  Set expr = 1 && 0
  Set expr = a && b
  Set expr = a > b

• An **object expression** produces a object reference as a result:
5.1.2.1 Logical Expressions

Logical expressions use logical operators, numeric relational operators, and string relational operators. They evaluate expressions and result in a Boolean value: TRUE (1) or FALSE (0). Logical expressions are most commonly used with:

- The IF command
- The $SELECT function
- Postconditional Expressions

When an expression contains logical operators, Caché ObjectScript evaluates all operands, even when the Boolean result is known before evaluating all operands.

In the following routine, all functions are executed even though the first function returns a FALSE which automatically makes the result of the entire expression FALSE.

```
logexp ; comment
If $$one() & $$two() {
    Write !,"Expression is TRUE."
} Else {
    Write !,"Expression is FALSE."
}
one()
Write !,"one"
Quit 0
two()
Write !,"two"
Quit 1
```

5.1.3 Assignment

Within ObjectScript the SET command is used along with the assignment operator (=) to assign a value to a variable. The right-hand side of an assignment command is an expression:

```
Set value = 0
Set value = a + b
```

Within ObjectScript it is also possible to use certain functions on the left-hand side of an assignment command:

```
Set pies = "apple,banana,cherry"
Write "Before: ",pies,!

// set the 3rd comma-delimited piece of pies to coconut
Set $Piece(pies,",",3) = "coconut"
Write "After: ",pies
```
5.2 Arithmetic Operators

The arithmetic operators interpret their operands as numeric values and produce numeric results.

5.2.1 Unary Positive Operator (+)

The unary positive operator (+) gives its single operand a numeric interpretation. If its operand has a string value, it converts it to a numeric value. It does this by sequentially parsing the characters of the string as a number, until it encounters an invalid character. It then returns whatever leading portion of the string was a well-formed numeric. For example:

Write + "32 dollars and 64 cents"        // 32

If the string has no leading numeric characters, the unary positive operator gives the operand a value of zero. For example:

Write + "Thirty-two dollars and 64 cents" // 0

The unary positive operator has no effect on numeric values. It does not alter the sign of either positive or negative numbers. For example:

Set x = -23
Write " x: ", x,! // -23
Write "+x: ",+x,! // -23

5.2.2 Unary Negative Operator (-)

The unary negative operator (-) reverses the sign of a numerically interpreted operand. For example:

Set x = -60
Write " x: ", x,! // -60
Write "-x: ",-x,! // 60

If its operand has a string value, the unary negative operator interprets it as a numeric value before reversing its sign. This numeric interpretation is exactly the same as that performed by the unary positive operator, described above. For example:

Set x = -23
Write "-"32 dollars and 64 cents" // -32
Caché ObjectScript gives the unary negative operator precedence over the binary arithmetic operators. Caché ObjectScript first scans a numeric expression and performs any unary negative operations. Then, Caché ObjectScript evaluates the expression and produces a result.

In the following example, Caché ObjectScript scans the string and encounters the numeric value of 2 and stops there. It then applies the unary negative operator to the value and uses the concatenate operator (_) to concatenate the value “Rats” from the second string to the numeric value.

\[ \text{Write } -"2Cats"_"Rats" \] // -2Rats

To return the absolute value of a numeric expression, use the $ZABS function.

### 5.2.3 Addition Operator (+)

The addition operator produces the sum of two numerically interpreted operands. It uses any leading valid numeric characters as the numeric values of the operands and produces a value that is the sum of the numeric value of the operands.

The following example performs addition on two numeric literals:

\[ \text{Write } 2936.22 + 301.45 \] // -3237.67

The following example performs addition on two locally defined variables:

Set \( x = 4 \)
Set \( y = 5 \)
Write "\( x + y = \)", \( x + y \) // 9

The following example performs string arithmetic on two operands that have leading digits, adding the resulting numerics:

\[ \text{Write } "4 Motorcycles" + "5 bicycles" \] // 9

The following example illustrates that leading zeros on a numerically evaluated operand do not affect the results the operator produces:

\[ \text{Write } "007" + 10 \] // 17

### 5.2.4 Subtraction Operator (-)

The subtraction operator produces the difference between two numerically-interpreted operands. It interprets any leading, valid numeric characters as the numeric values of the operand and produces a value that is the remainder after subtraction.

The following example performs subtraction on two numeric literals:
Write 2936.22 - 301.45 // 2634.77

The following example performs subtraction on two locally defined variables:

Set x = 4
Set y = 5
Write "x - y = ", x - y // -1

The following example performs string arithmetic on two operands that have leading digits, subtracting the resulting numerics:

Write "8 apples" - "4 oranges" // 4

If the operand has no leading numeric characters, Caché ObjectScript assumes its value to be zero. For example:

Write "8 apples" - "four oranges" // 8

5.2.5 Multiplication Operator (*)

Binary Multiply produces the product of two numerically interpreted operands. It uses any leading numeric characters as the numeric value of the operands and produces a result that is the product.

The following example performs multiplication on two numeric literals:

Write 9 * 5.5 // 49.5

The following example performs multiplication on two locally defined variables:

Set x = 4
Set y = 5
Write x * y // 20

The following example performs string arithmetic on two operands that have leading digits, multiplying the resulting numerics:

Write "8 apples" * "4 oranges" // 32

If an operand has no leading numeric characters, Binary Multiply assigns it a value of zero.

Write "8 apples"*"four oranges" // 0
5.2.6 Division Operator (/)

Binary Divide produces the result of dividing two numerically-interpreted operands. It uses any leading numeric characters as the numeric value of the operands and products a result that is the quotient.

The following example performs division on two numeric literals:

Write 9 / 5.5 // 1.636363636363636364

The following example performs division on two locally defined variables:

Set x = 4
Set y = 5
Write x / y // .8

The following example performs string arithmetic on two operands that have leading digits, dividing the resulting numerics:

Write "8 apples" / "4 oranges" // 2

If the operand has no leading numeric characters, Binary Divide assumes its value to be zero. For example:

Write "eight apples" / "4 oranges" // 0
// "8 apples"/"four oranges" generates a <DIVIDE> error

Note that the second of these operations is invalid. Dividing a number by zero is not allowed. Caché ObjectScript returns a <DIVIDE> error message.

5.2.7 Exponentiation Operator (**)

The Exponentiation Operator produces the exponentiated value of the left operand raised to the power of the right operand. The left and right operands can be positive or negative numbers, or zero. However, if the left operand is zero (0), the right operand must be a positive number. Zero raised to the power of any positive number is zero. Attempting to raise zero to the power of zero or to a negative number results in an <ILLEGAL VALUE> error.

The following examples performs exponentiation on two numeric literals:

Write "9 ** 2 = ",9 ** 2,! // 81
Write "9 ** -2 = ",9 ** -2,! // .01234567901234567901
Write "9 ** 2.5 = ",9 ** 2.5,! // 242.9999999994422342

The following example performs exponentiation on two locally defined variables:

Set x = 4, y = 3
Write "x ** y = ",x ** y,! // 64
The following example performs string arithmetic. Exponentiation uses any leading numeric characters as the values of the operands and produces a result.

```objectscript
Write "4 apples" ** "3 oranges" // 64
```

If an operand has no leading numeric characters, Exponentiation assumes its value to be zero. The following example demonstrates how to use exponentiation to find the square root of a number.

```objectscript
Write 256 ** .5 // 16
```

### 5.2.8 Integer Divide Operator ( \ )

The Integer Divide operator produces the integer result of the division of the left operand by the right operand. It does not return a remainder, and does not round up the result.

The following example performs integer division on two integer operands. Caché ObjectScript does not return the fractional portion of the number:

```objectscript
Write "355 \ 113 = ", 355 \ 113 // 3
```

The following example performs string arithmetic. Integer Divide uses any leading numeric characters as the values of the operands and produces an integer result.

```objectscript
Write "8 Apples" \ "3.1 oranges" // 2
```

If an operand has no leading numeric characters, Caché ObjectScript assumes its value to be zero. If you attempt integer division with a zero-valued divisor, Caché ObjectScript returns a <DIVIDE> error.

### 5.2.9 Modulo Operator (#)

The Modulo operator produces the value of an arithmetic modulo operation on two numerically interpreted operands. When the two operands are positive, then the modulo operation is the remainder of the left operand integer divided by the right operand.

The following examples perform modulo operations on numeric literals, returning the remainder:

```objectscript
Write "37 # 10 = ",37 # 10,! // 7
Write "12.5 # 3.2 = ",12.5 # 3.2,! // 2.9
```

The following example performs string arithmetic. When operating on strings, they are converted to numeric values (according the values described in the section Variable Typing and
Conversion) prior to the applying of the modulo operator. Hence, the following two expressions are equivalent:

```plaintext
Write "8 apples" # "3 oranges",! // 2
Write 8 # 3 // 2
```

Since Caché evaluates a string with no leading numeric characters to zero, a right operand of this kind yields a DIVIDE error.

### 5.3 Logical Comparison Operators

The logical comparison operators compare the values of their operands and return a boolean value: TRUE (1) or FALSE (0).

#### 5.3.1 Unary Not

Unary Not inverts the truth value of the boolean operand. If the operand is TRUE (1), Unary Not gives it a value of FALSE (0). If the operand is FALSE (0), Unary Not gives it a value of TRUE (1).

For example, the following statements produce a result of FALSE (0):

```plaintext
SET x=0
WRITE x
```

While the following statements produces a result of TRUE (1).

```plaintext
SET x=0
WRITE 'x
```

Unary Not with a comparison operator inverts the sense of the operation it performs. It effectively inverts the result of the operation. For example, the following statement displays a result of FALSE (0):

```plaintext
WRITE 3>5
```

But, the following statement displays a result of TRUE (1):

```plaintext
WRITE 3'>5
```
5.3.2 Precedence and Logical Operators

Because Caché ObjectScript performs a strict left-to-right evaluation of operators, logical comparisons involving other operators must use parentheses to group operations to achieve the desired precedence. For example, you would expect the logical Binary Or (!) test in the following program to return TRUE (1):

```objectscript
SET x=1,y=0
WRITE x=1!y=0  // Returns FALSE (surprise!)
```

However, to properly perform this logical comparison, you must use parentheses to nest the other operations. The following example gives the expected results:

```objectscript
SET x=1,y=0
WRITE (x=1)!(y=0)  // Returns TRUE as expected
```

5.3.3 Binary And

Binary And tests whether both its operands have a truth value of TRUE (1). If both operands are TRUE (that is, have nonzero values when evaluated numerically), Caché ObjectScript produces a value of TRUE (1). Otherwise, Caché ObjectScript produces a value of FALSE (0).

There are two forms to Binary And: & and &&.

- The & operator evaluates both operands and returns a value of FALSE (0) if either operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).
- The && operator evaluates the left operand and returns a value of FALSE (0) if it evaluates to a value of zero. Only if the left operand is non-zero does the && operator then evaluates the right operand. It returns a value of FALSE (0) if the right operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).

The following examples evaluate two nonzero-valued operands as TRUE and produces a value of TRUE (1).

```objectscript
Set A=-4,B=1
Write A&B // TRUE (1)
```

returns 1.

```objectscript
Set A=-4,B=1
Write A&&B // TRUE (1)
```

returns 1.
The following examples evaluate one true and one false operand and produces a value of FALSE (0).

```
Set A=1,B=0
Write "A = ",A,!  
Write "B = ",B,!
Write "A&B = ",A&B, ! // FALSE (0)
Set A=1,B=0
Write "A&&B = ",A&&B, ! // FALSE (0)
```

both return FALSE (0).

The following examples show the difference between the “&” operator and the “&&” operator. In these examples, the left operand evaluates to FALSE (0) and the right operand is not defined. The “&” and “&&” operators respond differently to this situation:

- The “&” operator attempts to evaluate both operands, and fails with an <UNDEFINED> error.

```
Set A=0
Write A&B
```

- The “&&” operator evaluates only the left operand, and produces a value of FALSE (0).

```
Set A=0
Write A&&B // FALSE (0)
```

5.3.3.1 Not And (NAND)

You can specify the Boolean Not And (NAND) operation by using the Unary Not operator with the Binary And (&) operator in either of the following equivalent formats:

```
operand '& operand
'(operand & operand)
```

The Not And operation reverses the truth value of the & Binary And applied to both operands. It produces a value of TRUE (1) when either or both operands are false. It produces a value of FALSE when both operands are TRUE.

The && Binary And operator cannot be prefixed with a Unary Not operator: the format “&&” is not supported. However, the following format is supported:

```
'(operand && operand)
```

The following example performs two equivalent Not And operations. Each evaluates one FALSE (0) and one TRUE (1) operand and produces a value of TRUE (1).
Set A=0,B=1
Write !,A\&B   // Returns 1
Write !,'(A&B) // Returns 1

The following example performs a Not And operation by performing a && Binary And operation and then using a Unary Not to invert the result. The && operation tests the first operand and, because the boolean value is FALSE (0), && does not test the second operand. The Unary Not inverts the resulting boolean value, so that the expression returns TRUE (1):

Set A=0
Write !,'(A&&B)   // Returns 1

5.3.4 Binary Or

Binary Or produces a result of TRUE (1) if either operand has a value of TRUE or if both operands have a value of TRUE (1). Binary Or produces a result of FALSE (0) only if both operands are FALSE (0).

There are two forms to Binary Or: ! (exclamation point) and || (two vertical bars).

- The ! operator evaluates both operands and returns a value of FALSE (0) if both operand evaluates to a value of zero. Otherwise it returns a value of TRUE (1).
- The || operator evaluates the left operand. If the left operand evaluates to a non-zero value, the || operator returns a value of TRUE (1) without evaluating the right operand. Only if the left operand evaluates to zero does the || operator then evaluate the right operand. It returns a value of FALSE (0) if the right operand also evaluates to a value of zero. Otherwise it returns a value of TRUE (1).

The following examples evaluate two TRUE (non-zero) operands, apply the Binary Or to them, and produces a TRUE result:

Set A=5,B=7
Write "A|B = ",A||B,
Set A=5,B=7
Write "A|||B = ",A|||B,

both return TRUE (1).

The following examples evaluate one false and one true operand, apply the Binary Or to them, and produces a TRUE result:

Set A=0,B=7
Write "A|B = ",A||B,
Set A=0,B=7
Write "A|||B = ",A|||B,

both return TRUE (1).
The following examples evaluate two false operands and produces a result with a value of FALSE.

```
Set A=0,B=0
Write "A!B = ",A!B,!
Set A=0,B=0
Write "A||B = ",A||B,!
```

both return FALSE (0).

### 5.3.4.1 Not Or (NOR)

You can produce a Not Or (NOR) operation by using Unary Not with the ! Binary Or in either of the following equivalent formats:

```
operand '!' operand
'(operand ! operand)
```

The Not Or operation produces a result of TRUE (1) if both operands have values of FALSE. The Not Or operation produces a result of FALSE (0) if either operand has a value of TRUE or if both operands are TRUE.

The || Binary Or operator cannot be prefixed with a Unary Not operator: the format “'||” is not supported. However, the following format is supported:

```
'(operand | | operand)
```

The following Not Or examples evaluate two false operands and produce a TRUE result.

```
Set A=0,B=0
Write "A'!B = ",A'!B   // Returns 1
Set A=0,B=0
Write "'(A!B) = ",'(A!B)   // Returns 1
```

The following Not Or examples evaluate one TRUE and one false operand and produce a result of FALSE.

```
Set A=0,B=1
Write "A'!B = ",A'!B   // Returns 0
Set A=0,B=1
Write "'(A!B) = ",'(A!B)   // Returns 0
```

The following Not Or (||) example evaluates the left operand, and because it is TRUE (1) does not evaluate the right operand. The Unary Not inverts the resulting boolean value, so the expression returns FALSE (0).

```
Set A=1
Write "'(A||B) = ",'(A||B)   // Returns 0
```
5.4 String Operators

The string operator interprets its operands as strings and returns a string value. The single string operator is the Binary Concatenate operator ( _ ).

5.4.1 Binary Concatenate

You use Binary Concatenate to combine string literals, expressions, and variables. It takes the form:

operand_operand

Binary Concatenate produces a result that is a string composed of the right operand appended to the left operand. Binary Concatenate gives its operands no special interpretation. It treats them as string values.

The following example concatenates two strings:

Write "High"_"chair"

returns “Highchair”.

The following example concatenates two numeric literals and a string:

Write 609_-"_24

returns 609-24.

The following example concatenates two strings and the null string:

Set A="ABC"_"_"DEF
Write A

returns “ABCDEF”.

The null string has no effect on the length of a string. You can concatenate an infinite number of null strings to a string.
5.5 Numeric Relational Operators

There are two types of relational operators: string relational operators and numeric relational operators. Numeric relational operators use the numeric values of the operands to produce a Boolean result.

5.5.1 Binary Less Than

The Binary Less Than operator tests whether its left operand is numerically less than its right operand. Caché ObjectScript evaluates both operands numerically and returns a Boolean result of TRUE (1) if the left operand has a lesser numeric value than its right operand. Caché ObjectScript returns a Boolean result of FALSE (0) if the left operand has an equal or greater numeric value than the right operand. For example:

Write 9 < 6
returns 0.

Write 22 < 100
returns 1.

5.5.2 Binary Greater Than

Binary Greater Than tests whether the left operand is numerically greater than the right operand. Caché ObjectScript evaluates the two operands numerically and produces a result of TRUE (1) if the left operand is numerically larger than the right operand. It produces a result of FALSE (0) if the left operand is numerically equal to or smaller than the right operand. For example:

Write 15 > 15
returns 0.

Write 22 > 100
returns 0.
### 5.5.3 Greater Than or Equal To

You can produce a Greater Than or Equal To operation by using a Unary NOT operator (') with Binary Less Than (<). The two operators used together reverse the truth value of the Binary Less Than. Caché ObjectScript produces a result of TRUE (1) when the left operand is numerically greater than or equal to the right operand. It produces a result of FALSE (0) when the left operand is numerically less than the right operand.

You can express the Greater Than or Equal To operation in either of the following ways:

\[ \text{operand}_{\text{A}} '\lt \text{operand}_{\text{B}} \]

\[^(\text{operand}_{\text{A}} < \text{operand}_{\text{B}})\]

### 5.5.4 Less Than or Equal To

You can produce a Less Than or Equal To operation by using a Unary NOT operator (') with Binary Greater Than (>). The two operators used together reverse the truth value of the Binary Greater Than. Caché ObjectScript produces a result of TRUE (1) when the left operand is numerically less than or equal to the right operand. It produces a result of FALSE (0) when the left operand is numerically greater than the right operand.

You can express the Less Than or Equal To operation in either of the following ways:

\[ \text{operand}_{\text{A}} '\gt \text{operand}_{\text{B}} \]

\[^(\text{operand}_{\text{A}} > \text{operand}_{\text{B}})\]

The following example tests two variables in a Less Than or Equal To operation. Because both variables have an identical numerical value, the result is TRUE.

```objectscript
Set A="55",B="55"
Write A>B
```

returns 1.

### 5.6 String Relational Operators

String relational operators use the string interpretation of the operands to produce a Boolean result. You can precede any of the string relational operators with the NOT logical operator (') to obtain the negation of the logical result.
5.6.1 Binary Equals

Binary Equals tests two operands for string equality. When you apply Binary Equals to two strings, Caché ObjectScript returns a result of TRUE (1) if the two operands are identical strings with identical character sequences and no intervening characters, including spaces; otherwise it returns a result of FALSE (0). For example:

Write "SEVEN"="SEVEN"

returns TRUE (1).

Binary Equals does not imply any numeric interpretation of either operand. For example, the following statement produces a value of FALSE (0), even though the two operands are numerically identical:

Write "007"="7"

returns FALSE (0).

You can use Binary Equals to test for numeric equality if both operands have a numeric value. For example:

Write 007=7

returns TRUE (1).

You can also force a numeric conversion by using the Unary Arithmetic Positive. For example:

Write +"007"="7"

returns TRUE (1).

5.6.1.1 Not Equals

You can specify a Not Equals operation by using the Unary Not operator with Binary Equals. You can express the Not Equals operation in two ways:

operand '!= operand
'(operand = operand)

Not Equals reverses the truth value of the Binary Equals operator applied to both operands. If the two operands are not identical, the result is TRUE (1). If the two operands are identical, the result is FALSE (0).
5.6.2 Binary Contains

Binary Contains tests whether the sequence of characters in the right operand is a substring of the left operand. If the left operand contains the character string represented by the right operand, the result is TRUE (1). If the left operand does not contain the character string represented by the right operand, the result is FALSE (0). If the right operand is the null string, the result is always TRUE.

The following example tests whether L contains S. Because L does contain S, the result is TRUE (1).

```objectscript
Set L="Steam Locomotive", S="Steam"
Write L[S
```

returns TRUE (1).

The following example tests whether P contains S. Because the character sequence in the strings is different (a period in P and an exclamation point in S), the result is FALSE (0).

```objectscript
Set P="Let's play.", S="Let's play!"
Write P[S
```

returns FALSE (0).

5.6.2.1 Does Not Contain

You can produce a Does Not Contain operation by using the Unary Not character with Binary Contains in either of the following equivalent formats:

```
operand A [ operand B
'(operand A [ operand B)
```

The Does Not Contain operation returns TRUE if operand A does not contain the character string represented by operand B and FALSE if operand A does contain the character string represented by operand B. For example,

```objectscript
Set P="Beatles", S="Mick Jagger"
Write P'[S
```

returns 1.
5.6.3 Binary Follows

Binary Follows tests whether the characters in the left operand come after the characters in the right operand in ASCII collating sequence. Binary Follows tests both strings starting with the left most character in each. The test ends when either:

- A character is found in the left operand that is different from the character at the corresponding position in the right operand.
- There are no more characters left to compare in either of the operands.

Caché ObjectScript returns a value of TRUE if the first unique character in the left operand has a higher ASCII value than the corresponding character in the right operand (that is, if the character in the left operand comes after the character in the right operand in ASCII collating sequence.) If the right operand is shorter than the left operand, but otherwise identical, Caché ObjectScript also returns a value of TRUE.

Caché ObjectScript returns a value of FALSE if any of the following conditions prevail:

- The first unique character in the left operand has a lower ASCII value than the corresponding character in the right operand.
- The left operand is identical to the right operand.
- The left operand is shorter than the right operand, but otherwise identical.

The following example tests whether the string LAMPOON follows the string LAMP in ASCII collation order. The result is TRUE.

```objectscript
Write "LAMPOON"]"LAMP"
```

returns TRUE (1).

The following example tests whether the string in B follows the string in A. Because BO follows BL in ASCII collation sequence, the result is TRUE.

```objectscript
Set A="BLUE",B="BOY"
Write B]A
```

returns TRUE (1).

5.6.3.1 Not Follows

You can produce a Not Follows operation by using the Unary Not operator with Binary Follows in either of the following equivalent formats:
If all characters in the operands are identical or if the first unique character in operand A has a lower ASCII value that the corresponding character in operand B, the Not Follows operation returns a result of TRUE. If the first unique character in operand A has a higher ASCII value that the corresponding character in operand B, the Not Follows operation returns a result of FALSE.

In the following example, because C in CDE does follow A in ABC, the result is FALSE.

Write "CDE"']"ABC",!
Write '("CDE"]"ABC")

returns FALSE (0).

5.6.4 Binary Sorts After

Binary Sorts after tests whether the left operand sorts after the right operand in numeric subscript collation sequence. In numeric collation sequence, the null string collates first, followed by canonic numbers in numeric order with negative numbers first, zero next, and positive numbers, followed lastly by non-numeric values.

The Binary Sorts After operator returns a TRUE (1) if the first operand sorts after the second and a FALSE (0) if the first operand does not sort after the second. For example:

Write 122]]2

returns TRUE (1).

Write "LAMPOON"]]"LAMP"

returns TRUE (1).

5.6.4.1 Not Sorts After

You can produce a Not Sorts After operation by using the Unary Not operator with Binary Sorts After in either of the following equivalent formats:

operand A ']] operand B
') (operand A ]] operand B)

If operand A is identical to operand B or if operand B sorts after operand A, then Caché ObjectScript returns a result of TRUE. If operand A sorts after operand B, Caché ObjectScript returns a result of FALSE.
5.7 Pattern Matching

The Pattern Match operator tests whether the characters in its left operand are correctly specified by the pattern in its right operand.

For example, the following tests if a string, ssn, contains a valid U.S. Social Security Number (3 digits, a hyphen, 2 digits, a hyphen, and 4 digits):

```
Set Match = SSN ?3N1"-"2N1"-"4N
```

Note that there is no whitespace following the ? operator. Whitespace within the pattern must be within a quoted string and is interpreted as being part of the pattern.

The general format for a pattern match operation is as follows:

```
operand?pattern
```

<table>
<thead>
<tr>
<th>operand</th>
<th>The string whose characters you want to test for a pattern.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>Can be one of the following: A sequence of one or more patatoms; A indirect reference that evaluates to a sequence of one or more patatoms</td>
</tr>
</tbody>
</table>

A patatom can be one of the following:

- repcount patcharacter
- repcount stringliteral
- repcount alternation

<table>
<thead>
<tr>
<th>repcount</th>
<th>A repeat count—the exact number of instances to be matched. The repcount can evaluate to a number or to the period character (.). Use the period to specify any number of instances.</th>
</tr>
</thead>
<tbody>
<tr>
<td>patcharacter</td>
<td>A pattern code.</td>
</tr>
<tr>
<td>stringliteral</td>
<td>A string literal enclosed in double quotes.</td>
</tr>
<tr>
<td>alternation</td>
<td>A set of patatom sequences to choose from to pattern match a segment of the operand string (provides logical OR capability in pattern specifications).</td>
</tr>
</tbody>
</table>
Use a string literal enclosed in double quotes in a pattern if you want to match a specific character or characters. In other situations, use the special pattern codes provided by Caché ObjectScript. The following table shows the available pattern codes and their meanings:

### Pattern Match Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Matches any of the 52 ASCII upper- and lower-case alphabetic characters (that is, A through Z and a through z).</td>
</tr>
<tr>
<td>C</td>
<td>Matches any of the 33 ASCII control characters (ASCII codes 0 through 31 and 127).</td>
</tr>
<tr>
<td>E</td>
<td>Matches any character in the text.</td>
</tr>
<tr>
<td>L</td>
<td>Matches any of the 26 ASCII lower-case alphabetic characters (that is, a through z).</td>
</tr>
<tr>
<td>N</td>
<td>Matches any of the 10 ASCII numeric characters (that is, 0 through 9).</td>
</tr>
<tr>
<td>P</td>
<td>Matches any of the 33 ASCII punctuation characters (ASCII codes 32 through 47, 58 through 64, 91 through 96, and 123 through 126).</td>
</tr>
<tr>
<td>U</td>
<td>Matches any of the 26 ASCII upper-case alphabetic characters (that is, A through Z).</td>
</tr>
<tr>
<td>ZFWCHARZ</td>
<td>Matches any of the characters in the Japanese ZENKAKU character set (ASCII codes 12354 through 12542 except for 12447, 12448).</td>
</tr>
<tr>
<td>ZHWKATAZ</td>
<td>Matches any of the characters in the Japanese HANKAKU Kana character set (ASCII codes 65393 through 65439).</td>
</tr>
</tbody>
</table>

The pattern codes are not case sensitive. You can specify them in either upper case or lower-case. For example, ?5N is equivalent to ?5n.

The Pattern Match operator produces a result of TRUE (1) when the pattern correctly specifies the pattern of characters in the operand. It produces a result of FALSE (0) if the pattern does not correctly specify the pattern of characters in the operand.

The Pattern Match operator differs from the Binary Contains ([) operator. The Binary Contains operator returns TRUE (1) even if only a substring of the left-hand operand matches the right-hand operand. Also, Binary Contains expressions do not provide the range of options available with the Pattern Match operator. In Binary Contains expressions, you can use only a single string as the right-hand operand, without any special codes.

For example, assume that variable var2 contains the value “abc”. Consider the following Pattern Match expression:
Set match = var2?2L

This sets match to FALSE (0) because var2 contains three lowercase characters, not just two.

You can extend the scope of a pattern match by specifying:

- How many times a pattern can occur
- Multiple patterns
- A combination pattern
- An indefinite pattern
- An alternating pattern

### 5.7.1 Specifying How Many Times a Pattern Can Occur

To define a range for the number of times that pattern can occur in the target operand, use the form:

\[ n.n \]

The first \( n \) defines the lower limit for the range of occurrences; the second \( n \) defines the upper limit.

For example, assume that the variable var3 contains the string “ABABAB”. In the following expression, 1.4 indicates that from one to four occurrences of “AB” are recognized:

\[ \text{Set match = var3?1.4"AB"} \]

The expression returns a result of TRUE (1) even though var3 contains only three occurrences of “AB”.

As another example, consider the following expression:

\[ \text{Set match = var3?1.6A} \]

This expression checks to see whether var3 contains from one to six upper-case alphabetic characters. A result of FALSE (0) is returned only if var3 contains zero or more than six characters.

If you omit either \( n \), Caché ObjectScript supplies a default. The default for the first \( n \) is zero (0). The default for the second \( n \) is any number. Consider the following example:

\[ \text{Set match = var3?1."AB"} \]
This example returns a result of TRUE (1) as long as \textit{var3} contains at least one occurrence of the pattern string “AB”.

### 5.7.2 Specifying Multiple Patterns

To define multiple patterns, you can combine \textit{n} and pattern in a sequence of any length. Consider the following example:

```
Set match = date?2N1”/”2N1”/”2N
```

This expression checks for a date value in the format mm/dd/yy. The string “4/27/98” would return FALSE (0) because the month has only one digit. To detect both one and two digit months, you could modify the expression as:

```
Set match = date?1.2N1”/”2N1”/”2N
```

Now the first pattern match (1.2N) accepts either 1 or 2 digits. It uses the optional period (.) to define a range of acceptable occurrences as described in the previous section.

### 5.7.3 Specifying a Combination Pattern

To define a combination pattern, use the form:

```
PatternPattern
```

With a combination pattern, the sequence consisting of the first \textit{pattern} followed by the second \textit{pattern} is checked against the target operand. For example, consider the following expression:

```
Set match = value?3N.4L
```

This expression checks for a pattern in which three numeric digits are followed by zero to four lowercase alphabetic characters. The expression returns TRUE (1) only if the target operand contains exactly one occurrence of the combined pattern. For example, the string “345gfij” would qualify, but “345gfij276hkbc” would not.

### 5.7.4 Specifying an Indefinite Pattern

To define an indefinite pattern, use the form:

```
.pattern
```

With an indefinite pattern, the target operand is checked for an occurrence of \textit{pattern}, but any number of occurrences is accepted (including zero occurrences). For example, consider the expression:
Set match = value?.N

This expression returns TRUE (1) if the target operand contains zero, one, or more than one numeric character, and contains no characters of any other type.

### 5.7.5 Specifying an Alternating Pattern (Logical OR)

You use an alternation to specify that any one of a number of pattern sequences can be used to pattern match a particular segment of your operand string. It adds a logical OR capability to pattern matching that simplifies your patterns.

An alternation has the following syntax:

```
( patatom sequence {, patatom sequence }...)  
```

Thus, the following pattern returns TRUE (1) if `val` contains one occurrence of the letter “A” or one occurrence of the letter “B”.

```
Set match = value?1(1"A",1"B")
```

You can have nested alternation patterns, as in the following pattern match expression:

```
Set match = value?1.(1A,1N),1P)
```

For example, you may want to validate a U.S. telephone number. At a minimum, the phone number must be a 7-digit phone number with a hyphen (-) separating the third and fourth digits. For example:

```
nnn-nnnn
```

The phone number can also include a three-digit area code that must either have surrounding parentheses or be separated from the rest of the number by a hyphen. For example:

```
(nnn) nnn-nnnn
nnn-nnn-nnnn
```

The following pattern match expressions describe three valid forms of a U.S. telephone number:

```
Set match = phone?3N1"-"4N
Set match = phone?3N1"-"3N1"-"4N
Set match = phone?1"("3N1") "3N1"-"4N
```

Without an alternation, the following compound Boolean expression would be required to validate any form of U.S. telephone number.
Set match =
{
(phone?3N1"-"4N) ||
(phone?3N1"-"3N1"-"4N) ||
(phone?1"("3N1") "3N1"-"4N)
}

With an alternation, the following single pattern can validate any form of U.S. telephone number:

Set match = phone?.1(1"("3N1") "3N1"-")3N1"-"4N

The alternation in this example allows the area code component of the phone number to be satisfied by either 1"("3N1") " or 3N1"-". The alternation count range of 0 to 1 indicates that the operand phone can have 0 or 1 area code components.

Alternations with a repeat count greater than one (1) can produce many combinations of acceptable patterns. The following alternation matches the string shown and matches 26 other three-character strings.

Set match = "CAT"?3(1"C",1"A",1"T")

5.7.6 Using Incomplete Patterns

If a pattern match successfully describes only part of a string, then the pattern match returns a result of FALSE (0). That is, there cannot be any string left over when the pattern is exhausted. The following expression evaluates to a result of FALSE (0) because the pattern does not match the final “R”:

Set match = "RAW BAR"?.U1P2U

5.7.7 Multiple Pattern Interpretations

There can be more than one interpretation of a pattern as it is matched against an operand. For example, the following expression can be interpreted in two ways:

Set match = "/\\\\/A#####B$$$$"?.E1U.E

1. The first “.E” matches the substring “\\\\/”, the 1U matches the “A”, and the second “.E” matches the substring “#####B$$$$”.

2. The first “.E” matches the substring “\\\\/A#####”, the 1U matches the character “B”, , and the second “.E” matches the substring “$$$$$”.

As long as at least one interpretation of the expression is TRUE (1), then the expression has a value of TRUE.
5.7.8 Not Match Operator

You can produce a Not Match operation by using the Unary Not operator with Pattern Match:

.inspect(operand?pattern)

Not Match reverses the truth value of the Pattern Match. If the characters in the operand cannot be described by the pattern, then Not Match returns a result of TRUE (1). If the pattern matches all of the characters in the operand, then Not Match returns a result of FALSE (0).

5.8 Indirection

The Caché ObjectScript indirection operator (@) allows you to assign values indirectly to variables. Indirection is a technique that provides dynamic runtime substitution of part or all of a command line, a command, or a command argument by the contents of a data field. Caché performs the substitution before execution of the associated command.

Indirection should not be used with Caché object dot syntax. This is because dot syntax is parsed at macro compile time, not at runtime.

Although indirection can promote more economical and more generalized coding than would be otherwise available, it is never essential. You can always duplicate the effect of indirection by other means, such as by using the XECUTE command.

You should use indirection only in those cases where it offers a clear advantage. Indirection can have an impact on performance because Caché performs the required evaluation at runtime, rather than during the compile phase. Also, if you use complicated indirections, be sure to document your code clearly. Indirections can sometimes be difficult to decipher.

Indirection is specified by the indirection operator (@) and, except for subscript indirection, takes the form:

@variable

where variable identifies the variable from which the substitution value is to be taken. The variable can be an array node.

The following routine illustrates that indirection looks at the entire variable value to its right.
Caché recognizes five types of indirection:

- **Name indirection**
- **Pattern indirection**
- **Argument indirection**
- **Subscript indirection**
- **$TEXT argument indirection**

Which type of indirection is performed depends on the context in which the @variable occurs. Each type of indirection is described separately below.

Indirection cannot be used with dot syntax. This is because dot syntax is parsed at compile time, not at runtime.

### 5.8.1 Name Indirection

In name indirection, the indirection evaluates to a variable name, a line label, or a routine name. Caché substitutes the contents of variable for the expected name prior to executing the command.

When you use indirection to reference a named variable, the value of the indirection must be a complete global or local variable name, including any necessary subscripts. In the following example, Caché sets the variable B to the value of 6.

```ObjectScript
Set Y = "B", @Y = 6
```

When you use indirection to reference a line label, the value of the indirection must be a syntactically valid line label. In the following example, Caché sets D to:

- The value of the line label FIG if the value of N is 1.
- The value of the line label GO if the value of N is 2.
- The value of STOP in all other cases.
Later, Caché passes control to the label whose value was given to D.

```
B Set D = $SELECT(N = 1:"FIG",N = 2:"GO",1:"STOP")
; ...
LV GoTo @D
```

When you use indirection to reference a routine name, the value of the indirection must be a syntactically valid routine name. In the following example, name indirection is used on the GoTo command to supply the appropriate subroutine name. At execution time, the contents of variable loc are substituted for the expected name.

If the indirection operator were omitted, Caché would return a <NOLINE> error message (or GoTo a label called loc, if it existed).

```
Start
Read !,"Enter choice (1, 2, or 3): ",num
Set loc = "Choice"_num
GoTo @loc
Quit
Choice1
; ...
Choice2
; ...
Choice3
; ...
```

Name indirection can substitute only a name value. The second Set command in the following example returns an error message because of the context. When evaluating the expression to the right of the equal sign, Caché interprets @var1 as an indirect reference to a variable name, not a numeric value.

```
Set var1 = "5"
Set x = @var1*6
```

You can recast the example to execute correctly as follows:

```
Set var1 = "var2",var2 = 5
Set x = @var1*6
```

### 5.8.2 Pattern Indirection

Pattern indirection is a special form of indirection. The indirection operator replaces a pattern match. The value of the indirection must be a valid pattern. (Pattern matching is described under "Pattern Matching".) Pattern indirection is especially useful when you want to select several possible patterns and then use them as a single pattern.

In the following example, indirection is used with pattern matching to check for a valid U.S. Postal (ZIP) code. Such codes can take either a five-digit \((nnnnn)\) or a nine-digit \((nnnnnnnnn)\) form.
The first **Set** command sets the pattern for the five-digit form. The second **Set** command sets the pattern for the nine-digit form. The second **Set** command is executed only if the postconditional expression \(\$LENGTH(zip) = 10\) evaluates to TRUE (non-zero), which occurs only if the user inputs the nine digit form.

```objectscript
Getzip
    Set pat = "5N"
    Read !,"Enter your ZIP code (5 or 9 digits): ",zip
    Set:($LENGTH(zip)=10) pat = "5N1"-"4N"
    IF zip'?@pat {
        Write !,"Invalid ZIP code"
        GoTo Getzip
    }
```

The use of indirection with pattern matching is a convenient way to localize the patterns used in an application. In this case, you could store the patterns in separate variables and then reference them with indirection during the actual pattern tests. (This is also an example of name indirection.) To port such an application, you would have to modify only the pattern variables themselves.

### 5.8.3 Argument Indirection

In argument indirection, the indirection evaluates to one or more command arguments. By contrast, name indirection applies only to part of an argument.

To illustrate this difference, compare the following example with the example given under **Name Indirection**.

```objectscript
Start
    Set rout = "^Test1"
    Read !,"Enter choice (1, 2, or 3): ",num
    Set loc = "Choice"_num_rout
    GoTo @loc
    Quit
```

In this case, @loc is an example of argument indirection because it supplies the complete form of the argument (that is, \(^{label}\)routine). In the name indirection example, @loc is an example of name indirection because it supplies only part of the argument (the \(^{label}\) name, whose entry point is assumed to be in the current, rather than a separate, routine).

In the following example, the second **Set** command is an example of name indirection (only part of the argument, the name of the variable), while the third **Set** command is an example of argument indirection (the entire argument).

```objectscript
Set a = "var1",b = "var2 = 3*4"
Set @a = 5*6
Set @b
Write "a = ",a,!
Write "b = ",b,!
```
5.8.4 Subscript Indirection

Subscript indirection is an extended form of name indirection. In subscript indirection, the value of the indirection must be the name of a local or global array node. Subscript indirection is syntactically different than the other forms of indirection. Subscript indirection uses two indirection operators in the following format:

`@array@(subscript)`

Assume that you have a global array called `^client` in which the first-level node contains the client's name, the second-level node contains the client's street address, and the third-level node contains the client's city, state, and ZIP code. To write out the three nodes for the first record in the array, you can use the following form of the `Write` command:

`Write !,^client(1),!,^client(1,1),!,^client(1,1,1)`

When executed, this command might produce output similar to following:

John Jones
42 Arnold St.
Boston, MA 02745

To write out a range of records (say, the first 10), you could modify the code so that the `Write` is executed within a `FOR` loop. For example,

```_objectscript
FOR i = 1:1:10 {
  Write !,^client(i),!,^client(i,1),!,^client(i,1,1)}
```

As the `FOR` loop executes, the variable `i` is incremented by 1 and used to select the next record to be output.

While more generalized than the previous example, this is still very specialized code because it explicitly specifies both the array name and the number of records to output.

To transform this code into a more generalized form that would allow a user to list a range of records from any array (global or local) that stores name, street, and city information in three node levels, you could use subscript indirection as shown in the following example:
Start
Read !,"Output Name, Street, and City info.",!
Read !,"Name of array to access: ",name
Read !,"Global or local (G or L): ",gl
Read !,"Start with record number: ",start
Read !,"End with record number: ",end
IF (gl["L"]!(gl["l"]) {Set array = name}
ELSEIF (gl["G"]!(gl["g"]) {Set array = "^"_name}
Set x = 1,y = 1
FOR i = start:1:end {DO Output}
Quit
Output Write !,@array@(i)
Write !,@array@(i,x)
Write !,@array@(i,x,y)
Quit

The Write commands in the Output subroutine use subscript indirection to reference the requested array and the requested range of records.

In the evaluation of subscript indirection, if the instance of indirection refers to an unsubscripted global or local variable, the value of the indirection is the variable name and all characters to the right of the second Indirection operator, including the parentheses.

5.8.5 $TEXT Argument Indirection

As its name implies, $TEXT argument indirection is allowed only in the context of a $TEXT function argument. The value of the indirection must be a valid $TEXT argument.

You use $TEXT argument indirection primarily as a convenience to avoid multiple forms of indirection that produce the same result. For example, if the local variable LINE contains the entry reference " START^MENU," you can use name indirection to the line label and to the routine name to obtain the text for the line, as follows:

    Set LINETEXT = $TEXT(@P(LINE,"^",1)^@P(LINE,"^",2))

You can use $TEXT argument indirection to produce the same result in a simpler manner, as follows:

    Set LINETEXT = $T(@LINE)
6

Commands

The command is the basic unit of code in Caché ObjectScript programming. All of the common execution tasks are controlled by commands.

Caché ObjectScript commands are similar to English sentences. Just as an English sentence expresses a complete thought, the command expresses a complete action that Caché ObjectScript can perform. Consider the following:

```
Write "HELLO"
```

The word “Write” is a command. It specifies the action to perform. The Write command does exactly what its name implies: it writes to the principle device whatever you specify as its argument. In this case, Write writes the string “HELLO”.

ObjectScript includes commands that perform various kinds of operations. These include:

- Invoking code—Do; Quit; Job; and Xecute
- Assignment—Set; Kill; and New
- Flow control—If, ElseIf, and Else; For; and While and Do/While
- Input/Output—Write; Read; and general I/O commands

This chapter addresses these commands and a quick overview of other commands, as well as describing the principles of command arguments and postconditionals.
6.1 Command Arguments

Following a command keyword, there can be zero, one, or multiple arguments that specify the object(s) or the scope of the command. For example, consider the command:

\[ \text{Set } x = y \times 2 \]

This command uses one argument, “\( x = y \times 2 \)” as a whole.

\( \text{Set} \) requires the equals sign, which you must always specify.) The argument instructs \( \text{Set} \) to multiply the current value of user-defined variable \( y \) by 2 and store the result in user-defined variable \( x \).

If a command takes one or more arguments, you must include exactly one space between the command keyword and the first argument. For example:

\[ \text{Set } x = y \times 2 \]

Spaces can appear within the first argument, so long as the first character of that argument is separated from the command itself by exactly one space (as appears above).

If a command takes a postconditional expression, there must be no spaces between the command keyword and the postconditional, and there must be exactly one space between the postconditional and the beginning of the first argument. Thus, the following are all valid forms of the \textbf{Quit} command:

\[
\begin{align*}
\text{Quit } x+y \\
\text{Quit } x + y \\
\text{Quit:}x<0 \\
\text{Quit:}x<0 \ x+y \\
\text{Quit:}x<0 \ x + y
\end{align*}
\]

6.1.1 Multiple Arguments

Many commands allow you to specify multiple arguments. The delimiter for command arguments is the comma “,”. That is, you specify multiple arguments to a single command as a comma-separated list following the command. For example:

\[ \text{Set } x=2, y=4, z=6 \]

This command uses three arguments to assign values to the three specified variables. In this case, these multiple arguments are repetitive; that is, the command is applied independently to each argument in the order specified. It is equivalent to three separate \textbf{Set} commands.
In the command syntax provided in the command reference pages, arguments that can be repeated are followed by a comma and ellipsis: , . . . . The comma is a required character for the argument, and the ellipsis indicates that the preceding argument can be repeated multiple times.

No spaces are required between arguments, but multiple blank spaces can be used between arguments. These blank spaces have no effect on the execution of the command. Line breaks, tabs, and comments can also be included within or between command arguments with no effect on the execution of the command.

Many command arguments also accept *parameters* (not to be confused with function parameters described in Functions).

If a given argument can take parameters, the delimiter for the parameters is the colon “:”. The following sample command shows the comma used as the argument delimiter and the colon used as the parameter delimiter. In this example, there are two arguments, with four parameters for each argument.

```
View X:y:z:a,B:a:y:z
```

### 6.1.2 Argumentless Commands

Commands that do not take an argument are referred to as *argumentless* commands. For example, **Halt** is an argumentless command.

Some commands are optionally argumentless (for example, the argumentless **Do** command). In such cases, the argumentless command may have a slightly different meaning than the same command with an argument.

If you use an argumentless command at the end of the line, spaces are not required.

If you use an argumentless command on the same code line as other commands, you must place *two* (or more) spaces between the argumentless command and any command that follows it. For example:

```
Quit:x=10  Write "not 10 yet"
```

In this case, **Quit** is an argumentless command with a postconditional expression, and a minimum of two spaces is required between it and the next command.

### 6.1.2.1 Argumentless Commands and Curly Braces

The spacing rules for argumentless commands when used in command blocks delimited by curly braces are as follows:
An argumentless command that can take an argument requires two or more spaces between the command keyword and the opening curly brace. For example, the `For` command can either take an argument, or be an argumentless command:

```objectscript
For {
   Write "Still in an endless loop"
}
```

An argumentless command keyword that cannot take an argument does not require the two spaces between the command keyword and the opening curly brace. For example, the `Else` keyword of the `If` command is always argumentless:

```objectscript
If (found) {
   Write "Found"
}
Else {
   Write "Not found"
   Quit x+y
}
```

An argumentless command that is immediately followed by a closing curly brace does not require the two spaces, because the closing curly brace acts as a delimiter. For example, the following is a valid use of the argumentless `Quit`:

```objectscript
If (found) {
   Write "Found"
}
Else {
   Write "Not found"
   Quit
}
Write "Done"
```

### 6.2 Command Postconditional Expressions

When you specify a Caché ObjectScript command, you can append to it a *postconditional*. A postconditional is an optional expression that is appended to a command or (in some cases) a command argument and controls whether Caché executes that command or command argument. If the postconditional expression evaluates to TRUE (defined as non-zero), Caché executes the command or the command argument. If the postconditional expression evaluates to FALSE (defined as zero), Caché does not execute the command or command argument, and execution continues with the next command or command argument.

All Caché ObjectScript commands can take a postconditional expression, except the flow-of-control commands: `If`, ` ElseIf`, and `Else`; `For`, `While`, and `Do...While`. 
The Caché ObjectScript commands **Do** and **Xecute** can append postconditional expressions both to the command keyword and to their command arguments. A postconditional expression is always optional; for example, some of the command's arguments may have an appended postconditional while its other arguments do not.

If both a command keyword and one or more of that command's arguments specify a postconditionals, the keyword postconditional is evaluated first. Only if this keyword postconditional evaluates to TRUE are the command argument postconditionals evaluated. If a command keyword postconditional evaluates to FALSE, the command is not executed and program execution continues with the next command. If a command argument postconditional evaluates to FALSE, the argument is not executed and execution of the command continues with the next argument in left-to-right sequence.

### 6.2.1 Postconditional Syntax

To add a postconditional to a command, place a colon (:) and an expression immediately after the command keyword, so that the syntax for a command with a postconditional expression is:

```
Command:pc
```

where **Command** is the command keyword, the colon is a required literal character, and **pc** can be any valid expression.

A command postconditional must follow these rules:

- No spaces, tabs, line breaks, or comments are permitted between a command keyword and its postconditional, or between a command argument and its postconditional.

- No spaces, tabs, line breaks, or comments are permitted within a postconditional expression, unless the postconditional expression is enclosed in parentheses. If parentheses are used, no spaces, tabs, line breaks, or comments are permitted between the keyword (or command argument) and the colon, or between the colon and the open parenthesis.

- Spacing requirements following a postconditional expression are the same as those following a command keyword: there must be exactly one space between the last character of the keyword postconditional expression and the first character of the first argument; for argumentless commands, there must be two or more spaces between the last character of the postconditional expression and the next command on the same line, unless the postconditional is immediately followed by a close curly brace. (If parentheses are used, the closing parenthesis is treated as the last character of the postconditional expression.)
Note that a postconditional expression is not technically a command argument (though in the ObjectScript reference pages the explanation of the postconditional is presented as part of the Arguments section). A postconditional is always optional.

### 6.2.2 Evaluation of Postconditionals

Caché evaluates a postconditional expression as either True or False. Most commonly these are represented by 1 and 0, which are the recommended values. However, Caché performs postconditional evaluation on any value, evaluating it as False if it evaluates to 0 (zero), and True if it evaluates to a non-zero value.

- Caché evaluates as True any valid non-zero numeric value. It uses the same criteria for valid numeric values as the arithmetic operators. Thus, the following all evaluate to True: 1, “1”, 007, 3.5, -.007, 7.0, “3 little pigs”, $CHAR(49), 0_"1".
- Caché evaluates as False the value zero (0), and any non-numeric value, including a null string ("" ) or a string containing a blank space (" "). Thus, the following all evaluate to False: 0, -0.0, “A”, “.”, “$”, “The 3 little pigs”, $CHAR(0), $CHAR(48), "0_1".
- Standard Caché equivalence rules apply. Thus, the following evaluate to True: 0=0, 0="0", “a”=$CHAR(97), 0=$CHAR(48), and (" ")=$CHAR(32)). The following evaluate to False: 0="", 0=$CHAR(0), and

For example, the following **Write** command’s action depends on the value of the variable **count**:

```objectscript
Set count = 4
Write:count<5 "count is less than 5.",!
Set count = 6
Write:count>5 "count is greater than 5.",!
```

### 6.3 Invoking Code

This section describes the **Do** command and other commands related to it. These are:

- **Do**
- **Quit**
- **Job**
- **Xecute**
6.3.1 Do

To invoke any routine, function, procedure, or method in ObjectScript, use the **Do** command. In this sense, **Do** and **Set** serve as fundamental commands in Caché, since **Do** allows you to call other code and **Set** allows you to specify variable values. The basic syntax of **Do** is:

```
Do ^CodeToInvoke
```

where *CodeToInvoke* can be a Caché system routine or a user-defined routine. The caret character “^” must appear immediately before the name of the routine.

You can run procedures within a routine by referring to the label of the line (also called a tag) where the procedure begins within the routine. The label appears immediately before the circumflex. For example,

```
Set %X = 484
Do INT^%SQROOT
Write %Y
```

This code sets the value of the %X system variable to 484; it then uses **Do** to invoke the INT procedure of the Caché system routine **%SQROOT**, which calculates the square root of the value in %X and stores it in %Y. The code then displays the value of %Y using the **Write** command.

When invoking methods, **Do** takes as a single argument the entire expression that specifies the method. The form of the argument depends on whether the method is an instance or a class method. To invoke a class method, use the following construction:

```
Do ##class(PackageName.ClassName).ClassMethodName()
```

where *ClassMethodName* is the name of the class method that you wish to invoke, *ClassName* is the name of the class containing the method, and *PackageName* is the name of the package containing the class. The **##class()** construction is a required literal part of the code.

To invoke an instance method, you need only have a handle to the locally instantiated object:

```
Do InstanceName.InstanceMethodName()
```

where *InstanceMethodName* is the name of the instance method that you wish to invoke and *InstanceName* is the name of the instance containing the method.
6.3.2 Quit

The Quit command terminates execution of a code block, including a method. Without an argument, it simply causing the invoking block to exit; with an argument, the invoking block uses the argument of Quit as a return value.

6.3.3 Job

While Do runs code in the foreground, Job runs it in the background. This occurs independently of the current process, usually without user interaction. A jobbed process inherits all system defaults, except those explicitly specified.

6.3.4 Xecute

The Xecute command runs one or more Caché ObjectScript commands; it does this by evaluating the expression that it receives as an argument (and its argument must evaluate to a string containing Caché ObjectScript commands). In effect, each Xecute argument is like a one-line subroutine called by a Do command and terminated when the end of the argument is reached or a Quit command is encountered. After Caché executes the argument, it returns control to the point immediately after the Xecute argument.

6.4 Assignment Commands

To provide the necessary functionality for variable management, ObjectScript provides the following commands:

- Set
- Kill
- New

6.4.1 Set

The Set command assigns values to variables. It can assign a value to a single variable or to multiple variables at once.

To most basic syntax of Set is:
Set variable = expression

This sets the value of a single variable. It also involves several steps:

- ObjectScript evaluates the expression, determining its value (if possible). This step can generate errors, if the expression contains an undefined variable, invalid syntax (such as division by zero), or other errors.
- If the variable does not already exist, ObjectScript creates it. This is an example.
- Once the variable has been created, or if it already exists, ObjectScript sets its value to that of the expression.

To set the value for each of multiple variables, use the following syntax:

```
Set variable1 = expression1, variable2 = expression2, variable3 = expression3
```

To set multiple variables equal to a single expression use the following syntax:

```
Set (variable1,variable2,variable3)= expression
```

For example, to set the value of the Gender property of an instance of the Person class use the following code:

```
Set person.Gender = "Female"
```

where `person` is the object reference to the relevant instance of the Person class.

You can also set the Gender property of multiple Person objects at the same time:

```
Set (per1.Gender, per2.Gender, per3.Gender) = "Male"
```

where `per1`, `per2`, and `per3` are object references to three different instances of the Person class.

You can also use `Set` to invoke a method that returns a value. When invoking methods, `Set` allows you to set a variable, global reference, or property equal to the return value of an expression that specifies the method. The form of the argument depends on whether the method is an instance or a class method. To invoke a class method, use the following construction:

```
Set LocalVariable = ##class(PackageName.ClassName).ClassMethodName()
```

where `LocalVariable` is that being assigned the return value, `ClassMethodName` is the name of the class method that you wish to invoke, `ClassName` is the name of the class containing the method, and `PackageName` is the name of the package containing the class. The `##class()` construction is a required literal part of the code.
To invoke an instance method, you need only have a handle to the locally instantiated object:

```
Set LocalVariable = InstanceName.InstanceMethodName()
```

where `LocalVariable` is that being assigned the return value, `InstanceMethodName` is the name of the instance method that you wish to invoke, and `InstanceName` is the name of the instance containing the method.

If you need to use the value that a method, class or instance, returns, invoke it using `Set`.

### 6.4.2 Kill

The **Kill** command deletes variables from memory and can be used to delete them from disk. Its basic form is:

```
Kill expression
```

where `expression` is one or more variables to delete. The simplest forms of Kill, then, are:

- `Kill x`
- `Kill y, z`

A special form of **Kill**, called an “exclusive **Kill**”, deletes all local variables *except* those specified. To use an exclusive **Kill**, place its argument in parentheses. For example, if you have variables `x, y, and z`, you can delete `y, z, and any other local variables except `x` by invoking:

```
Kill (x)
```

Without any arguments, **Kill** deletes all local variables.

### 6.4.3 New

The **New** command initializes variables. In an application that uses procedures, use **New** to initialize variables for the entire application or a major subsystem of the application.

### 6.5 Flow Control Commands

In order to establish the logic of any code, there must be flow control. To that end, ObjectScript supports the following commands:

- **If, ElseIf, and Else**
6.5.1 If, ElseIf, and Else

In order to control the flow of a routine, there must be a way to conditionally run or bypass certain sections of code. There also must be a way to cause certain sections of code to run repeatedly. The basic mechanism for this functionality in Caché is the **If** command.

**If** takes an expression as an argument and determines if that expression is true or false. If true, then the code that follows the expression is executed; if false, the code is not executed. The code usually appears in a *code block* containing multiple commands. Code blocks are simply one or more lines of code contained in curly braces; there can be line breaks before and within the code blocks. Consider the following:

```ObjectScript
Read "Pick a number: ", x
If x = 78 {
   Write !,"It's 78!"
}
```

For shorter code blocks, a postconditional may be simpler to use:

```ObjectScript
Write:x=78 !,"It's 78!"
```

### 6.5.1.1 If, ElseIf, and Else

The **If** *construct* allows you to evaluate multiple conditions, and to specify what code is run based on the conditions. A construct, as opposed to a simple command, consists of a combination of one or more arguments, command keywords, and *code blocks*. The full syntax of the **If** construct is:

```ObjectScript
if if-condition { code } elseif elseif-condition {code} else {code}
```

**ElseIf** and **Else** are both optional, and there can be more than one **ElseIf**. **If** there are two conditions, the construct is:

```ObjectScript
if if-condition { code } else {code}
```

The following is an example of the **If** construct:
Read "Enter the number of equal-length sides in the polygon: ",x
If (x = 1) {
    Write !,"It's so far away that it looks like a point!"
} ElseIf (x = 2) {
    Write !,"I think that's a line, not a polygon."
} ElseIf (x = 3) {
    Write !,"It's a triangle!"
} ElseIf (x = 4) {
    Write !,"It's a square!"
} Else {
    Write !,"That's a lot of sides!"
}

6.5.2 For

You use the **For** construct to repeat sections of code. You can create a **For** loop based on numeric or string values.

Typically, **For** executes a code block zero or more times based on the value of a numeric control variable that is incremented or decremented at the beginning of each loop through the code. When the control variable reaches its end value, control exits the **For** loop; if there is no end value, the loop executes until it encounters a **Quit** command. When control exits the loop, the control variable maintains its value from the last loop executed.

The form of a numeric **For** loop is:

```
For ControlVariable = StartValue:IncrementAmount:EndValue {
    // code block content
}
```

All values can be positive or negative; spaces are permitted but not required around the equals sign and the colons. The code block following the **For** will repeat for each value assigned to the variable.

For example, the following **For** loop will execute five times:

```
Write "The first five multiples of 3 are:",!
For multiple = 3:3:15 {
    Write multiple,!
}
```

You can also use a variable to determine the end value. In the example below, a variable specifies how many iterations of the loop occur:
Set howmany = 4
Write "The first ",howmany," multiples of 3 are "
For multiple = 1:1:howmany {
    Write (multiple*3),", "
    If multiple = (howmany - 1) {
        Write "and "
    } If multiple = howmany {
        Write "and that's it!"
    }
}
Quit

Because this example uses *multiple*, the control variable, to determine the multiples of 3, it displays the expression *multiple*\(^3\). It also uses the *If* command to insert “and” before the last multiple.

**Note:** The *If* command in this example provides an excellent example of the implications of order of precedence in ObjectScript (order of precedence is always left to right with no hierarchy among operators). If the *If* expression were simply “*multiple* = howmany - 1”, without any parentheses or parenthesized as a whole, then the first part of the expression, “*multiple* = howmany”, would be evaluated to its value of False (0); the expression as a whole would then be equal to “0 - 1”, which is -1, which means that the expression will evaluate as true (and insert “and” for every case except the final iteration through the loop).

The argument of *For* can also be a variable set to a list of values; in this case, the code block will repeat for each item in the list assigned to the variable.

```
For b = "John", "Paul", "George", "Ringo" {
    Write !, "Was ", b, " the leader? "
    Read choice
}
```

You can specify the numeric form of *For* without an ending value by placing a *Quit* within the code block that triggers under particular circumstances and thereby terminates the *For*. This approach provides a counter of how many iterations have occurred and allows you to control the *For* using a condition that is not based on the counter's value. For example, the following loop uses its counter to inform the user how many guesses were made:

```
For i = 1:1 {
    Read !, "Capital of MA? ", a
    If a = "Boston" {
        Write ".\ldots did it in ", i, " tries"
        Quit
    }
}
```

If you have no need for a counter, you can use the argumentless *For*:
For {  
  Read !, "Know what? ", wh  
  Quit:(wh = "No!")  
  Write " That's what!"
}

Note: The section below contains information on the conditions for using For with Quit instead of Do/While or While.

6.5.3 While and Do/While

Two related flow control commands are While and Do/While commands, each of which loops over a code block and terminates based on a condition. The two commands differ in when they evaluate the condition: While evaluates the condition before the entire code block and Do/While evaluates the condition after the block. As with For, a Quit within the code block terminates the loop.

The syntax for the two commands is:

do {code} while condition
while condition {code}

The following example displays values in the Fibonacci sequence up to a user-specified value twice—first using Do/While and then using While:

fibonacci() PUBLIC { // generate Fibonacci sequences
  Read !, "Generate Fibonacci sequence up to where? ", upto  
  Set t1 = 1, t2 = 1, fib = 1  
  Write !
  Do {
    Write fib," ", set fib = t1 + t2, t1 = t2, t2 = fib
  }  
  While ( fib '>' upto )
  Set t1 = 1, t2 = 1, fib = 1
  Write !
  While ( fib '>' upto ) {
    Write fib," ", set fib = t1 + t2, t1 = t2, t2 = fib
  }  
}

The distinction between While, Do/While, and For is that While necessarily tests the control expression's value prior to executing the loop, Do/While necessarily tests the value after executing the loop, and For can test it anywhere within the loop. This means that if you have two parts to a code block, where execution of the second depends on evaluating the expression, the For construct is best suited; otherwise, the choice depends on whether expression evaluation should precede or follow the code block.
6.6 I/O Commands

ObjectScript input/output commands provide the basic functionality for getting data in and out of Caché. These are:

- **Write**
- **Read**
- **I/O commands**

### 6.6.1 Write

The **Write** command displays content. It can be used with or without an argument.

With an argument that is a valid expression, **Write** displays the value of that expression; the expression can be a literal, a local variable, a global reference, an object property, or the value returned by method or function. Without an argument the **Write** command displays the names and values of all local variables, but neither global references, properties, or the results of any code.

For example, to write the literal string “Hello” to the current output device, use the following:

```
Write "Hello"
```

With one or more variables, **Write** displays their value:

```
Set x = 4
Set y = 3
Write x,!, // 4
Write y,!, // 3
Write x,y,!, // 43
Write "x = ",x,!,"y = ",y // output on multiple lines
```

**Note:** Since **Write** simply concatenates the elements of its output, you need to add any formatting that you wish to display, as in the example above.

You can use **Write** to display values of local variables, globals, and properties:

```
Set x = 2
Write "x = ",x,!
Set ^x = 4
Write "^x = ",^x,!
Set per = ##class(Sample.Person).%OpenId(5)
Write "per.Name = ",per.Name,!
Kill x, ^x, per
```
You can also display the output of methods and functions. For methods, the syntax is:

```
Write instance.method()
```

where `instance` is the object in memory, `method` is its method, and the displayed value is the return value of the method. For instance, the `Sample.Person` class (which comes with Caché) has a method called `NinetyNine`, which returns 99. Here, the code for instance of `Person` called `MyPerson` is:

```
Set MyPerson = ##class(Sample.Person).%OpenId(3)
Write MyPerson.NinetyNine()
```

To invoke a routine or other ObjectScript code that returns a value, the syntax is:

```
Write $$label^routine()
```

where `label` is the label in the routine, `routine` is the name of the routine being invoked, and the leading “$$” and trailing “()” are required. For instance, suppose there is a subroutine with the label “Fun” in the routine “house”:

```
Fun()
Set place = "Funhouse"
Quit place
```

Here, invoking “Fun^house” would be in the form have the result of:

```
USER>Write $$Fun^house()
Funhouse
```

When invoking a subroutine with `Write`, the subroutine must `Quit` with a value and the subroutine's label must have trailing parentheses; otherwise, errors will result.

### 6.6.2 Read

The `Read` command allows you to accept and store input entered by the end user via the current input device. The `Read` command can have any of the following arguments:

```
Read format, string, variable
```

Where `format` controls where the user input area will appear on the screen, `string` will appear on the screen before the input prompt, and `variable` will store the input data.

The following format codes are used to control the user input area:
<table>
<thead>
<tr>
<th>Format Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Starts a new line.</td>
</tr>
<tr>
<td>#</td>
<td>Starts a new page. On a terminal it clears the current screen and starts at the top of a new screen.</td>
</tr>
<tr>
<td>?n</td>
<td>Positions at the nth column position where n is a positive integer.</td>
</tr>
</tbody>
</table>

### 6.6.3 Open, Use, and Close

For more sophisticated device handling, Caché offers a wealth of options. In short, you can take ownership of an open device with the Open command; specify the current device with the Use command; and close an open device with the Close command. This process as a whole is described in Using Caché I/O Devices.

### 6.7 Other Commands

Other commands and groups of commands include:

- Transaction-related commands, including **Lock**, **TStart**, **TCommit**, and **TRollBack**. See the [transaction processing](#) chapter for more information.

- Device-related commands, including **Open**, **Use**, and **Close**. See the document Using Caché I/O Devices for more information.

- Debugging-related commands, including **Break** and **GoTo**. See the [debugging](#) chapter for more information.

- The **Hang** command, which suspends execution for a specified time.
Functions

Functions are pieces of code which perform common tasks. Functions may either return a value or execute silently without returning a value. Caché supplies dozens of functions, which can perform mathematical operations, manipulation of strings and time values, and perform other actions.

System-supplied functions (sometimes known as “intrinsic” functions), which you cannot modify, are identifiable as they always begin with a single dollar sign ( “$” ); the Caché ObjectScript Language Reference describes each of the system-supplied functions. In addition to its system-supplied functions, ObjectScript also supports user-defined code (sometimes known as “extrinsic” code).

7.1 Function Syntax

To invoke a system-supplied function, use the form:

    $name(parameters)

where
Invoking Caché ObjectScript Functions

| name     | The name of the function. The preceding dollar sign ($) is required. Function names are case-insensitive, whether system-supplied or user-defined (method names are not). Many system function names have abbreviations; each of these is listed in the reference page for the function itself. |
| parameters | One or more values to be passed to the function. The parentheses are mandatory, even if the function has no parameters. The parameters are positional and must match the order of the parameters expected by the function. Multiple parameters are separated from each other by commas. Spaces are permitted anywhere in the parameter list. No spaces are permitted between name and the open parenthesis character. |

Function **parameters** represent specific values that you supply. All defined parameters are shown in each syntax section. Some of these parameters are required and some are optional. Optional arguments and parameters are indicated as such in the Parameters section of each reference page.

Also shown in many syntax sections are required punctuation characters used with some parameters. These include parentheses, commas, curly braces, colons, equal signs, and plus and minus signs.

Function parameters are values that the function uses to produce its result. You place function parameters within parentheses. The opening parenthesis must immediately follow the function keyword with no intervening spaces, as follows:

```
Write $CHAR(65)
```

Spaces and comments are permitted within the parentheses, as shown in the following example:

```
Write $CHAR( 65 /* this parameter is the letter A */)
```

Some functions take one parameter, some take several. When specifying more than one function parameter, you separate the parameters with commas. Spaces and comments are permitted between parameters, before or after the comma.

```
Write $FIND(X,Y)
```

For most functions, you cannot specify multiple instances of the same parameter. The exceptions are **$CASE**, **$CHAR**, and **$SELECT**.
User-Defined Code

Caché comes with many system-supplied functions (sometimes known as “intrinsic” functions), which are described in the Caché ObjectScript Language Reference. As in other languages, ObjectScript allows you to create named code blocks that you can invoke directly. Such blocks are known as procedures. Strictly speaking, in ObjectScript terminology, a code block that is a procedure has a specific syntax and structure. By default, methods are procedures.

The syntax of a procedure definition:

```
ProcedureName(Parameters) [PublicVariables]
{
    // code...
    Quit ReturnValue
}
```

The elements of the procedure, above called `ProcedureName`, are:

- Parameters (zero or more) — These can be of any type and, as is typical of ObjectScript, you do not need to declare their types when you define the procedure. By default, they are passed by value (not by reference). Unless otherwise specified, their scope is local to the procedure. For more information on parameters generally, see the section “Procedure Parameters.”

- References to public variables (zero or more) — These, too, can be of any type. The procedure can both reference and set such a variable's value. For more information on public variable references, see the section “Procedure Variables.”

- Declaration that the procedure is public (optional) — By default, procedures are private, which means that you can only call them from elsewhere in the same routine (in ObjectScript terminology, a routine is a file containing one or more procedures or other user-defined code blocks). You can also create procedures that are public, using the
PUBLIC keyword after the procedure name. Public procedures can be called from other routines or methods. For more information on public and private procedures, see “Public and Private Procedures.”

- Code — The code in a procedure has all the features available in Caché ObjectScript. Procedure code can also include Basic and Java.

- Return value (optional) — This is the value that the procedure returns, and, must be a standard ObjectScript expression. Flow control within a procedure can specify various return values using computed expression values, multiple Quit statements, or both.

Note: For those familiar with versions of Caché prior to version 5 (and code written for those versions), procedures represent an advance from coding that was previously available with subroutines and user-defined functions. Procedure parameters are automatically local in scope within the procedure; they do not require a New command to ensure that they do not overwrite other values. Also, the explicit declaration of public variables allows you to refer to global variables within an application, such as a bank-wide interest rate; it also allows you to create and set values for variables within the procedure that are available to the rest of an application.

Procedures are a particular kind of ObjectScript routine.

Caché also provides a large number of system-supplied functions, all of which are described in the Caché ObjectScript Language Reference; these are sometimes known as intrinsic functions. Calls to system functions are identified by a “$” prefix.

8.1 Procedures, Routines, Subroutines, Functions, and Methods: What Are They?

This chapter describes how to implement your own code using procedures, which are the recommended form for implementing user-defined functionality. Caché documentation as a whole talks about procedures, routines, subroutines, functions, and methods. Though all these entities share features, each has its own characteristics. Most notably, applications that pre-exist Caché 5 often include subroutines and functions that are not implemented as procedures. For that reason, it is important to provide a brief overview of all of these different structures for implementing your own code.
The most flexible, most powerful, and recommended form of named, user-defined code block is the procedure. The features of a procedure includes that it:

- Can be private or public.
- Can accept zero or more arguments.
- Automatically maintains any variables created within it as local in scope.
- Can refer to and alter variables outside it.
- Can return a value of any type or no return value.

By contrast:

- A subroutine is always public and cannot return a value.
- A function is always public, requires explicit declaration of local variables (and, otherwise, overwrites external variables), and must have a return value.
- By default, a method is a procedure that is specified as part of a class definition and that you can invoke on one or more objects or on a class. If you explicitly declare it a function, it is then a function with all the accompanying characteristics; this is not recommended.
- A routine is a Caché ObjectScript program. It can include one or more procedures, subroutines, and functions, as well as any combination of the three.

Note: ObjectScript also supports a related form of user-defined code through its macro facility.

8.1.1 Routines

A routine is a callable block of user-written code that is a Caché ObjectScript program. A routine performs commonly-needed operations. Its name is determined by the name of the .MAC file that you choose for saving it. Depending on if a routine returns a value, you can invoke a routine with one or both of the following sets of syntax:

```objectscript
DO ^RoutineName
SET x = ^RoutineName
```

Generally, routines serve as containers for subroutines, methods, and procedures.

The routine is identified by a label (also referred to as a tag) at the beginning of the block of code. This label is the name of the routine. This label is (usually) followed by parentheses which contain a list of parameters to be passed from the calling program to the routine.
When you save a routine to a file, the file name cannot include the underscore ("_"), hyphen ("-"), or semicolon (";") characters; names that include such characters are not valid.

### 8.1.2 Subroutines

A subroutine is a named block of code within a routine. Typically, a subroutine begins with label and ends with a **Quit** statement. It can accept parameters and does not return a value. To invoke a subroutine, use the following syntax:

```plaintext
DO Subroutine^Routine
```

where *Subroutine* is a code block within the *Routine* file (Routine.MAC).

The form of a subroutine is:

```plaintext
Label(parameters) // comment
// code
Quit // note that Quit has no arguments
```

For more details on subroutines, see the section below on **subroutines** as legacy code.

If you enclose the code and **Quit** statement within curly braces, the subroutine is a procedure and can be treated as such.

### 8.1.3 Functions

A function is a named block of code within a routine. Typically, a function begins with label and ends with a **Quit** statement. It can accept parameters and can also return a value. To invoke a routine, there are two valid forms of the syntax:

```plaintext
DO Function^Routine
$$Function()
```

where *Function* is a code block within the *Routine* file (Routine.MAC).

The form of a subroutine is:

```plaintext
Label(parameters) // comment
// code
Quit ReturnValue
```

For more details on functions, see the section below on **functions** as legacy code.

If you enclose the code and **Quit** statement within curly braces, the function is a procedure and can be treated as such.
8.2 Procedures in Detail

As in other languages, a procedure is a series of ObjectScript commands (a section of a larger routine) that accomplishes a specific task. Similar to constructs such as `If`, the code of a procedure is contained within curly braces.

Procedures allow you to define each variable as either public or private. For example, the following procedure, is called “MyProc”:

```objectscript
MyProc(x,y) [a,b] PUBLIC {
  Write "x + y = ", x + y
}
```

defines a public procedure named “MyProc” which takes two parameters, `x` and `y`. It defines two public variables, `a` and `b`. All other variables used in the procedure (in this case, `X` and `Y`) are private variables.

By default, procedures are private, which means that you can only call them from elsewhere in the same routine. You can also create procedures that are public, using the `Public` keyword after the procedure name. Public procedures can be called from other routines.

Procedures need not have defined parameters. To create procedures with parameters, place a parenthesized list of variables immediately after the label.

8.2.1 Invoking Procedures

To invoke a procedure, either issue a `Do` command that specifies the procedure, or call it as a function using the “$$” syntax. You can control whether a procedure can be invoked from any program (public), or only from the program in which it is located (private). If invoked with `Do`, a procedure does not return a value; if invoked as a function call, a procedure returns a value. The “$$” form provides the most functionality, and is generally the preferred form.

8.2.1.1 Using the $$ Prefix

You can invoke a user-defined function in any context in which an expression is allowed. A user-defined function call takes the form:

```
$$name([param[ ,...]])
```

where:

- `name` specifies the name of the function. Depending on where the function is defined, name can be specified as:
- **label** is a line label within the current routine.
- **label^routine** is a line label within the named routine that resides on disk.
- **^routine** is a routine that resides on disk. The routine must contain only the code for the function to be performed.

- **param** specifies the values to be passed to the function. The supplied parameters are known as the *actual parameter list*. They must match the *formal parameter list* defined for the function. For example, the function code may expect two parameters, with the first being a numeric value and the second being a string literal. If you specify the string literal for the first parameter and the numeric value for the second, the function may yield an incorrect value or possibly generate an error. Parameters in the formal parameter list always have **New** invoked by the function. See the NEW command. Parameters can be passed by value or by reference. See Parameter Passing. If you pass fewer parameters to the function than are listed in the function's formal parameter list, parameter defaults are used (if defined); if there are no defaults, these parameters remain undefined.

### 8.2.1.2 Using the Do Command

You can invoke a user-defined function using the **Do** command. (You cannot invoke a system-supplied function using the **Do** command.) A function invoked using **Do** does not return a value. That is, the function must generate a return value, but the **Do** command ignores this return value. This greatly limits the use of **Do** for invoking user-defined functions.

To invoke a user-defined function using **Do**, you issue a command in the following syntax:

```
Do label(param[,...])
```

The **Do** command calls the function named **label** and passes it the parameters (if any) specified by **param**. Note that the $$ prefix is not used, and that the parameter parentheses are mandatory. The same rules apply for specifying the **label** and **param** as when invoking a user-defined function using the $$ prefix.

A function must always return a value. However, when a function is called with **Do**, this returned value is ignored by the calling program.

### 8.2.2 Procedure Syntax

Procedure syntax:

```
label([param[=default]][,...]) [[pubvar[,...]]] [access] { code }
```
Invoking syntax:

Do label([param[, ...]])

or

command $$label([param][, ...])

where

<table>
<thead>
<tr>
<th>label</th>
<th>The procedure name. A standard label. It must start in column one. The parameter parentheses following the label are mandatory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>param</td>
<td>A variable for each parameter expected by the procedure. These expected parameters are known as the <em>formal parameter list</em>. The parameters themselves are optional (there may be none, one, or more than one <em>param</em>) but the parentheses are mandatory. Multiple <em>param</em> values are separated by commas. Parameters may be passed to the formal parameter list by value or by reference.</td>
</tr>
<tr>
<td>default</td>
<td>An optional default value for the <em>param</em> preceding it. You can either provide or omit a default value for each parameter. A default value is applied when no actual parameter is provided for that formal parameter, or when an actual parameter is passed by reference and the local variable in question does not have a value. This default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string (<code>””</code>) as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, Caché issues a &lt;PARAMETER&gt; error.</td>
</tr>
<tr>
<td>pubvar</td>
<td>Public variables. An optional list of public variables used by the procedure and available to other routines and procedures. This is a list of variables both defined within this procedure and available to other routines and defined within another routine and available to this procedure. If specified, <em>pubvar</em> is enclosed in square brackets. If no <em>pubvar</em> is specified, the square brackets may be omitted. Multiple <em>pubvar</em> values are separated by commas. All variables not declared as public variables are private variables. Private variables are available only to the current invocation of the procedure. They are undefined when the procedure is invoked, and destroyed when the procedure is exited with a <strong>Quit</strong>. If the procedure calls any code outside of that procedure, the private variables are preserved, but are unavailable until the call returns to the procedure. All % variables are always public, whether or not they are listed here. The list of public variables can include one or more of the <em>param</em> specified for this routine.</td>
</tr>
</tbody>
</table>
An optional keyword that declares whether the procedure is public or private. There are two available values: PUBLIC, which declares that this procedure can be called from any routine. PRIVATE, which declares that this procedure can only be called from the routine in which it is defined. PRIVATE is the default.

A block of code, enclosed in curly braces. The opening curly brace (\{\}) must be separated from the characters preceding and following it by at least one space or a line break. The closing curly brace (\}) must not be followed by any code on the same line; it can only be followed by blank space or a comment. The closing curly brace can be placed in column one. This block of code is only entered by the label.

You cannot insert a line break between a command and its arguments.

Each procedure is implemented as part of a routine; each routine can contain multiple procedures.

In addition to standard ObjectScript syntax, there are special rules governing routines. A line in a routine can have a label at the beginning (also called a tag), ObjectScript code, and a comment at the end; but all of these elements are optional.

InterSystems recommends that the first line of a routine have a label matching the name of the routine, followed by a tab or space, followed by a short comment explaining the purpose of the routine. If a line has a label, you must separate it from the rest of the line with a tab or a space. This means that as you add lines to your routine using Caché Studio, you either type a label and a tab/space, followed by ObjectScript code, or you skip the label and type a tab or space, followed by ObjectScript. So in either case, every line must have a tab or space before the first command.

To denote a single-line comment use either a double forward-slash (//) or a semicolon (;). If a comment follows code, there must be a space before the slashes or semicolon; if the line contains only a comment, there must be a tab or space before the slashes or semicolon. By definition, there can be no line break within a single-line comment; for a multi-line comment, mark the beginning of the comment with “/\*” and the end with “\*/”.

### 8.2.3 Procedure Variables

Variables used within procedures are automatically private to that procedure. You don't have to declare them as such. To share some of these variables with procedures that this procedure calls, pass them as parameters to the other procedures.

You can also declare public variables. These are available to all procedures; those that this procedure calls and those that called this procedure. A relatively small number of variables
should be defined in this way, to act as environmental variables for an application. To define public variables, list them in square brackets following the procedure name and its parameters.

SAMPLES>Write
SAMPLES>Do ^publicvarsexample
setting a
setting b
setting c
The sum is: 6
SAMPLES>Write
a=1
b=2
SAMPLES>

The publicvarsexample.mac code:

```
publicvarsexample
  ; examples of public variables
  ;
  Do proc1() ; call a procedure
  Quit ; end of the main routine
  ;
proc1() [a, b]
  ; a private procedure
  ; "c" and "d" are private variables
  {
  Write !, "setting a" Set a = 1
  Write !, "setting b" Set b = 2
  Write !, "setting c" Set c = 3
  Set d = a + b + c
  Write !, "The sum is: ", d
  }
```

8.2.3.1 Public versus Private Variables

Within a procedure, local variables may be either “public” or “private”. The public list (pubvar) declares which variable references in the procedure are added to the set of public variables; all other variable references in the procedure are to a private set seen only by the current invocation of the procedure.

Private variables are undefined when a procedure is entered, and they are destroyed when a procedure is exited with a Quit.

When code within a procedure calls any code outside of that procedure, the private variables are restored upon the return to the procedure. The called procedure or routine has access to public variables (as well as its own private ones.) Thus, pubvar specifies both the public variables seen by this procedure and the variables used in this procedure that are capable of being seen by a routine that the procedure calls.

If the public list is empty, then all variables are private. In this case, the square brackets are optional.
Variables whose name starts with the “%” character are typically variables used by the system or for some special purpose. All such % variables are implicitly public. They can be listed in the public list (for documentation purposes) but this is not necessary.

### 8.2.3.2 Private Variables versus Variables Created with New

Note that private variables are not the same as variables newly created with `New`. If a procedure wants to make a variable directly available to other procedures or subroutines that it calls, then it must be a public variable and it must be listed in the public list. If it is a public variable being introduced by this procedure, then it makes sense to perform a `New` on it. That way it will be automatically destroyed when we `Quit` the procedure, and also it protects any previous value that public variable may have had. For example, the code:

```objectscript
MyProc(x,y) [name] {
  New name
  Set name="John"
  Do xyz^abc
  Quit
}
```

enables procedure “xyz” in routine “abc” to see the value “John” for `name`, because it is public. Invoking the `New` command for `name` protects any public variable named “name” that may already have existed when the procedure “MyProc” was called.

The `New` command doesn't affect private variables; it only works on public variables. Within a procedure, it is illegal to specify `New X` or `New (X)` if `X` is not listed in the public list and `X` is not a % variable.

### 8.2.3.3 Making Formal List Parameters Public

If a procedure has a formal list parameter, (such as “x” or “y” in `MyProc(x,y)`) that is needed by other procedures it calls, then the parameter should be listed in the public list.

Thus,

```objectscript
MyProc(x,y) [x] {
  Do abc^rou
}
```

makes the value of `x`, but not `y`, available to the routine “abc^rou”.

### 8.2.4 Public and Private Procedures

A procedure can be “public” or “private”. A private procedure can only be called from within the routine in which the procedure is defined, whereas a public procedure can be called
from any routine. If the PUBLIC and PRIVATE keywords are omitted, the default is “private”.

For instance,

MyProc(X,Y) PUBLIC { }

defines a public procedure, while

MyProc(X,Y) PRIVATE { }

and

MyProc(X,Y) { }

both define a private procedure.

8.2.5 Procedure Parameters

An important feature of procedures is their support for parameter passing. This is the mechanism by which you can pass values (or variables) to a procedure as arguments. It is not, of course, required, such as to generate a random number or to return the system date in a format other than the default format. To set up parameter passing, specify:

• An actual parameter list on the function call.

• A formal parameter list on the function definition

When Caché executes a user-defined function, it maps the parameters in the actual list, by position, to the corresponding parameters in the formal list. For example, the value of the first parameter in the actual list is placed in the first variable in the formal list; the second value is placed in the second variable; and so on. The matching of these parameters is done by position, not name. Thus, the variables used for the actual parameters and the formal parameters are not required to have (and usually should not have) the same names. The function accesses the passed values by referencing the appropriate variables in its formal list.

If there are more variables in the formal list than there are parameters in the actual list, and a default value is not provided for each, the extra variables are left undefined. Your function code should include appropriate If tests to make sure that each function reference provides usable values.

When passing parameters to a user-defined function, you can use passing by value or passing by reference. You can mix passing by value and passing by reference within the same function call. See Parameter Passing.
Parameter passing can be done by reference or by value.

- System-supplied functions—these can be passed parameters by value only.
- User-defined functions—these can be passed parameters by value or by reference.
- Subroutines can be passed parameters by value or by reference.
- Procedures can be passed parameters by value or by reference.

### 8.2.5.1 Passing By Value

To pass by value, specify an actual value, an expression, or an unsubscripted local variable name in the actual parameter list. In the case of an expression, Caché first evaluates the expression and then passes the resulting value. In the case of a variable name, Caché passes the variable's current value. Note that a specified variable name must already exist and must have a value.

Caché implicitly creates and declares any non-public variables used within a procedure, so that already-existing variables with the same name in calling code are not overwritten. It places the existing values for these variables (if any) on the program stack. When it invokes the `Quit` command, Caché executes an implicit `Kill` command for each of the formal variables and restores their previous values from the stack.

In the following example, the `Set` commands use three different forms to pass the same value to the referenced `Cube` procedure.

```objectscript
Do Start()
Start() PUBLIC {
  Set var1=6
  Set a=$$Cube(6)
  Set b=$$Cube(2*3)
  Set c=$$Cube(var1)
  Quit 1
}
Cube(num) PUBLIC {
  Set result = num*num*num
  Quit result
}
```
8.2.5.2 Passing By Reference

To pass by reference, specify a local variable name or the name of an unsubscripted array in the form:

```
.name
```

With passing by reference, a specified variable or array name does not have to exist prior to the function reference. Passing by reference is the only way you can pass an array name to a function.

The period preceding the variable or array name is required. It distinguishes passing by reference from passing by value using a variable name. If you specify a numeric literal that starts with a decimal point, the Caché ObjectScript compiler correctly interprets it as a value rather than a reference.

In passing by reference, each variable or array name in the actual list is bound to the corresponding variable name in the function's formal list. Passing by reference provides an effective mechanism for two-way communication between the referencing routine and the function. Any change that the function makes to a variable in its formal list is also made to the corresponding by-reference variable in the actual list. This also applies to the **Kill** command. If a by-reference variable in the formal list is killed by the function, the corresponding variable in the actual list is also killed.

If a variable or array name specified in the actual list does not already exist, the function reference treats it as undefined. If the function assigns a value to the corresponding variable in the formal list, the actual variable or array is also defined with this value.

The following example uses passing by reference to achieve two-way communication between the referencing routine and the function through the variable result. When the function is executed, result is created and bound to `z` in the function's formal list. The calculated value is assigned to `z` and passed back to the referencing routine in result. Note that `num` and `powr` are passed by value, not reference. This is an example of mixing passing by value and passing by reference.

```
Start ; Raise an integer to a power.
Read !,"Integer=",num Quit:num=""
Read !,"Power=",powr Quit:powr=""
Set output=$$Expo(num,powr,.result)
Write !,"Result=",output
Goto Start
Expo(x,y,z)
  Set z=x
  For i=1:1:y {Set z=z*x}
  Quit z
```

```
8.2.6 Procedure Code

The body of code between the braces is the procedure code, and it differs from traditional ObjectScript code in the following ways:

- A procedure can only be entered at the procedure tag. Access to the procedure through “tag+offset” syntax is not allowed.

- Any tags in the procedure are private to the procedure and can only be accessed from within the procedure. The PRIVATE keyword can be used on tags within a procedure, although it is not required. The PUBLIC keyword cannot be used on tags within a procedure—it yields a syntax error. Even the system function $TEXT cannot access a private tag by name, although $TEXT does support tag+offset using the procedure tag name.

- Duplicate tags are not permitted within a procedure but, under certain circumstances, are permitted within a routine. Specifically, duplicate tags are permitted within different procedures. Also, the same tag can appear within a procedure and elsewhere within the routine in which the procedure is defined. For instance, the following three occurrences of “TAG1” are permitted:

Rou1 // Rou1 routine
Proc1(X,Y) {
TAG1 // TAG1 within the proc1 procedure within the Rou1 routine
}

Proc2(A,B,C) {
TAG1 // TAG1 within the Proc2 procedure (local, as with previous TAG1)
}

TAG1 // TAG1 that is part of Rou1 and neither procedure

- If the procedure contains a Do or user-defined function without a routine name, it refers to a tag within the procedure, if one exists. Otherwise, it refers to a tag in the routine but outside of the procedure.

- If the procedure contains a Do or user-defined function with a routine name, it always identifies a line outside of the procedure. This is true even if that name identifies the routine that contains the procedure. For example:

ROU1 ;
PROC1(X,Y) {
Do TAG1^ROU1
TAG1 ;
}
TAG1 ; The Do calls this tag

- If a procedure contains a Goto, it must be to a private tag within the procedure. You cannot exit a procedure with a Goto.
"Tag+offset" syntax is not supported within a procedure, with a few exceptions:

- \textdollar TEST supports tag+offset from the procedure tag.
- Goto tag+offset is supported in direct mode lines from the procedure tag as a means of returning to the procedure following a Break or error.
- The ZBreak command supports a specification of tag+offset from the procedure tag.
- The \$TEST state in effect when the procedure was called is restored upon the Quit for the procedure.
- The \"}\" that denotes the end of the procedure can be in any character position on the line, including the first character position. Code can precede the \"}\" on the line, but cannot follow it on the line.
- An implicit Quit is present just before the closing brace.
- Indirection and Xecute commands behave as if they are outside of a procedure.

8.2.7 Indirection and Xecute Commands within Procedures

Name indirection, argument indirection, and Xecute commands that appear within a procedure are not executed within the scope of the procedure. Thus, Xecute acts like an implied Do of a subroutine that is outside of the procedure.

Indirection and Xecute only access public variables. As a result, if indirection or an Xecute references a variable $X$, then it references the public variable $X$ regardless of whether or not there is also a private $X$ in the procedure. For example:

\begin{verbatim}
Set x=\"set a=3\" XECUTE x ; sets the public variable a to 3
Set x=\"tag1\" Do @x ; accesses the public subroutine tag1
\end{verbatim}

Similarly, a reference to a tag within indirection or an Xecute is to a tag outside of the procedure. Hence Goto @A is not supported within a procedure, since a Goto from within a procedure must be to a tag within the procedure.

Other parts of the documentation contain more detail on indirection and the Xecute command.

8.2.8 Error Traps within Procedures

If an error trap gets set from within a procedure, it needs to be directly to a private tag in the procedure. (This is unlike in legacy code, where it can contain \"+offset\" or a routine name. This rule is consistent with the idea that executing an error trap essentially means unwinding the stack back to the error trap and then executing a Goto.)
If an error occurs inside a procedure, $ZERROR gets set to the procedure “tag+offset”, not to a private “tag+offset”.

To set an error trap, the normal $ZTRAP or $ETRAP is used, but the value must be a literal. For instance:

Set $ZT = "abc"
// sets the error trap to the private tag "abc" within this block

For more information on error traps, see the chapter of this document on Error Processing.

### 8.3 Legacy User-Defined Code

Prior to the addition of procedures to Caché, there was support for user-defined code in the form of subroutines and functions (which themselves can now be implemented as procedures). These legacy entities are described here, primarily to help explicate already-written code; their ongoing use is not recommended.

#### 8.3.1 Subroutines

#### 8.3.1.1 Syntax

**Routine syntax:**

```objectscript
label [ ( param [ = default ] [ , ... ] ) ]
/* code */
Quit
```

**Invoking syntax:**

```objectscript
Do label [ ( param [ , ... ] ) ]
```

or

```objectscript
Goto label
```

<table>
<thead>
<tr>
<th>label</th>
<th>The name of the subroutine. A standard label. It must start in column one. The parameter parentheses following the label are optional. If specified, the subroutine cannot be invoked using a Goto call. Parameter parentheses prevent code execution from “falling through” into a subroutine from the execution of the code that immediately precedes it. When Caché encounters a label with parameter parentheses (even if they are empty) it performs an implicit Quit, ending execution rather than “falling through.”</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>param</strong></td>
<td>The parameter value(s) passed from the calling program to the subroutine. A subroutine invoked using the <strong>Goto</strong> command cannot have <strong>param</strong> values, and must not have parameter parentheses. A subroutine invoked using the <strong>Do</strong> command may or may not have <strong>param</strong> values. If there are no <strong>param</strong> values, empty parameter parentheses may be specified or omitted. Specify a <strong>param</strong> variable for each parameter expected by the subroutine. The expected parameters are known as the <strong>formal parameter list</strong>. There may be none, one, or more than one <strong>param</strong>. Multiple <strong>param</strong> values are separated by commas. Caché automatically invokes <strong>New</strong> on the referenced <strong>param</strong> variables. Parameters may be passed to the formal parameter list by value or by reference.</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>An optional default value for the <strong>param</strong> preceding it. You can either provide or omit a default value for each parameter. A default value is applied when no actual parameter is provided for that formal parameter, or when an actual parameter is passed by reference and the local variable in question does not have a value. This default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string (&quot;&quot;) as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, Caché issues a &lt;PARAMETER&gt; error.</td>
</tr>
<tr>
<td><strong>code</strong></td>
<td>A block of code. This block of code is normally accessed by invoking the label. However, it can also be entered (or reentered) by calling another label within the code block or issuing a label + offset <strong>Goto</strong> command. A block of code can contain nested calls to other subroutines, functions, or procedures. It is recommended that such nested calls be performed using <strong>Do</strong> commands or function calls, rather than a linked series of <strong>Goto</strong> commands. This block of code is normally exited by an explicit <strong>Quit</strong> command; this <strong>Quit</strong> command is not always required, but is a recommended coding practice. You can also exit a subroutine by using a <strong>Goto</strong> to an external label.</td>
</tr>
</tbody>
</table>

### 8.3.1.2 Description

A subroutine is a block of code identified by a label found in the first column position of the first line of the subroutine. Execution of a subroutine most commonly completes by encountering an explicit **Quit** statement.

A subroutine is invoked by either the **Do** command or the **Goto** command.
User-Defined Code

- A **Do** command executes a subroutine and then resumes execution of the calling routine. Thus, when Caché encounters a **Quit** command in the subroutine, it returns to the calling routine to execute the next line following the **Do** command.

- A **Goto** command executes a subroutine but does not return control to the calling program. When Caché encounters a **Quit** command in the subroutine, execution ceases.

You can pass parameters to a subroutine invoked by the **Do** command; you cannot pass parameters to a subroutine invoked by the **Goto** command. You can pass parameters by value or by reference. See Parameter Passing.

The same variables are available to a subroutine and its calling routine.

A subroutine does not return a value.

### 8.3.2 Functions

As of Caché 5, a function, by default and recommendation, is a procedure. You can, however, continue to define a function that is not a procedure. This section describes such functions.

#### 8.3.2.1 Syntax

Non-procedure function syntax:

```
label([param [default] , ...])
  code
  Quit expression
```

Invoking syntax:

```
command $$label([param , ...])
```

or

```
Do label([param , ...])
```
<table>
<thead>
<tr>
<th>label</th>
<th>The name of the function. A standard label. It must start in column one. The parameter parentheses following the label are mandatory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>param</td>
<td>A variable for each parameter expected by the function. The expected parameters are known as the formal parameter list. There may be none, one, or more than one param. Multiple param values are separated by commas. Caché automatically invokes New for the referenced param variables. Parameters may be passed to the formal parameter list by value or by reference.</td>
</tr>
<tr>
<td>default</td>
<td>An optional default value for the param preceding it. You can either provide or omit a default value for each parameter. A default value is applied when no actual parameter is provided for that formal parameter, or when an actual parameter is passed by reference and the local variable in question does not have a value. This default value must be a literal: either a number, or a string enclosed in quotation marks. You can specify a null string (&quot;&quot;&quot;) as a default value. This differs from specifying no default value, because a null string defines the variable, whereas the variable for a parameter with no specified or default value would remain undefined. If you specify a default value that is not a literal, Caché issues a &lt;PARAMETER&gt; error.</td>
</tr>
<tr>
<td>code</td>
<td>A block of code. This block of code can contain nested calls to other functions, subroutines, or procedures. Such nested calls must be performed using Do commands or function calls. You cannot exit a function's code block by using a Goto command. This block of code can only be exited by an explicit Quit command with an expression.</td>
</tr>
<tr>
<td>expression</td>
<td>The function's return value, specified using any valid Caché ObjectScript expression. The Quit command with expression is a mandatory part of a user-defined function. The value that results from expression is returned to the point of invocation as the result of the function.</td>
</tr>
</tbody>
</table>

**8.3.2.2 Description**

User-defined functions are described in this section. Calls to user-defined functions are identified by a “$$” prefix. (A user-defined function is also known as an extrinsic function.)

User-defined functions allow you to add functions to those supplied by Caché. Typically, you use a function to implement a generalized operation that can be invoked from any number of programs.
A function is always called from within a Caché command. It is evaluated as an expression and returns a single value to the invoking command. For example:

`Set x=$myfunc()`

### 8.3.2.3 Function Parameters

As a rule, user-defined functions use parameter passing. A function, however, can work without externally-supplied values. For example, you can define a function to generate a random number or to return the system date in a format other than the default format. Note that in these cases, too, you must supply the parameter parentheses in both the function definition and the function call, even though the parameter list is empty.

Parameter passing requires:
- An [actual parameter list](#) on the function call.
- A [formal parameter list](#) on the function definition.

When Caché executes a user-defined function, it maps the parameters in the actual list, by position, to the corresponding parameters in the formal list. For example, the value of the first parameter in the actual list is placed in the first variable in the formal list; the second value is placed in the second variable; and so on. The matching of these parameters is done by position, not name. Thus, the variables used for the actual parameters and the formal parameters are not required to have (and usually should not have) the same names. The function accesses the passed values by referencing the appropriate variables in its formal list.

If there are more variables in the formal list than there are parameters in the actual list, and a default value is not provided for each, the extra variables are left undefined. Your function code should include appropriate **If** tests to make sure that each function reference provides usable values.

When passing parameters to a user-defined function, you can use passing by value or passing by reference. You can mix passing by value and passing by reference within the same function call. See [Parameter Passing](#).

### 8.3.2.4 Return Value

The basic form for defining a user-defined function is as follows:

```objectscript
label ( parameters )
/* code */
Quit expression
```
The function must contain a **Quit** command followed by an expression. Caché terminates the execution of the function when it encounters the **Quit**, and returns the single value that results from the associated expression to the invoking program.

If you specify a **Quit** command without an expression, Caché issues an error.

### 8.3.2.5 Variables

The invoking program and the called function use the same set of variables, with the following special considerations.

- Caché executes an implicit **New** command for each parameter in the formal list. This is shown in the following example, where x is re-initialized when myfunc is invoked:

  ```ObjectScript
  mainprog
  Set x=7
  Set y=$$myfunc(99)
  myfunc(x)
  Write x
  Quit 66
  ```

- Caché saves the current value of the system variable $TEST when it enters the function and restores it when the function terminates. Any change in the $TEST value during execution of the function will be discarded when the function exits, unless you include code to explicitly save it by some other means.

### 8.3.2.6 Location of Functions

You can define a user-defined function within the routine that references it, or in a separate routine where multiple programs can reference it. Recommended practice is to use one routine to contain all your user-defined function definitions. In this way, you can easily locate any function definition to examine or update it.

### 8.3.2.7 Invoking a User-Defined Function

You can invoke a user-defined function using either the $$ prefix, or by using the Do command. The $$ form provides the most functionality, and is generally the preferred form.

**Using the $$ Prefix**

You can invoke a user-defined function in any context in which an expression is allowed. A user-defined function call takes the form:

```ObjectScript
$$name([param [,...]])
```

where
User-Defined Code

- **name** specifies the name of the function. Depending on where the function is defined, name can be specified as:
  - *label* — A line label within the current routine.
  - *label^routine* — A line label within the named routine that resides on disk.
  - ^routine — A routine that resides on disk. The routine must contain only the code for the function to be performed.

- **param** specifies the values to be passed to the function. The supplied parameters are known as the actual parameter list. They must match the formal parameter list defined for the function. For example, the function code may expect two parameters, with the first being a numeric value and the second being a string literal. If you specify the string literal for the first parameter and the numeric value for the second, the function may yield an incorrect value or possibly generate an error. Parameters in the formal parameter list always have New invoked by the function. See the NEW command. Parameters can be passed by value or by reference. See Parameter Passing. If you pass fewer parameters to the function than are listed in the function's formal parameter list, parameter defaults are used (if defined); if there are no defaults, these parameters remain undefined.

**Using the Do Command**

You can invoke a user-defined function using the Do command. (You cannot invoke a system-supplied function using the Do command.) A function invoked using Do does not return a value. That is, the function must generate a return value, but the Do command ignores this return value. This greatly limits the use of Do for invoking user-defined functions.

To invoke a user-defined function using Do, you issue a command in the following syntax:

\[
\text{Do} \text{ label}(\text{param}[ ,...])
\]

The Do command calls the function named *label* and passes it the parameters (if any) specified by *param*. Note that the $$ prefix is not used, and that the parameter parentheses are mandatory. The same rules apply for specifying the *label* and *param* as when invoking a user-defined function using the $$ prefix.

A function must always return a value. However, when a function is called with Do, this returned value is ignored by the calling program.
Caché ObjectScript supports macros, which provide the functionality for simple textual substitutions in routines. In their basic form, they are created with a \#Define statement. For instance, the following code creates a macro called MyMacro and makes it a substitution for the string “Hello, World!”

```
\#Define StringMacro "Hello, World!"
```

ObjectScript allows you to invoke a macro using the “$$” syntax, such as:

```
Write $$$StringMacro
```

which, in this case, displays the string “Hello, World!” Here is the entire sample:

```
\#Define StringMacro "Hello, World!"
Write $$$StringMacro
```

In addition to substitution macros, ObjectScript also supports conditional macros.

### 9.1 Defining Macros

Macros are one-line definitions of substitutions that can support many aspects of Caché ObjectScript functionality. The following macros demonstrate the various supported features. Supported functionality includes:

- String substitutions, as demonstrated above.
- Numeric substitutions:
  ```
  \#define NumberMacro 22
  ```
As is typical in ObjectScript, the definition of the numeric macro does not require quoting the number, while the string must be quoted in the string macro's definition.

- **Variable substitutions:**
  
  ```
  #define VariableMacro Variable
  ```

  Here, the macro name substitutes for the name of a variable that is already defined. If the variable is not defined, there is an <UNDEFINED> error.

- **Command and argument invocations:**
  
  ```
  #define CommandArgumentMacro(%Arg) Write %Arg,!
  ```

  Macro argument names must start with the “%” character, such as the %Arg argument above. Here, the macro invokes the Write command, which uses the %Arg argument.

- **Use of functions, expressions, and operators:**
  
  ```
  #define FunctionExpressionOperatorMacro ($ZDate(+$Horolog))
  ```

  Here, the macro as a whole is an expression whose value is the return value of the $ZDate function. $ZDate operates on the expression that results from the operation of the “+” operator on the system time, which the system variable $Horolog holds. As shown above, it is a good idea to enclose expressions in parentheses so that they minimize their interactions with the statements in which they are used.

- **References to other macros:**
  
  ```
  #define ReferenceOtherMacroMacro Write $$FunctionExpressionOperatorMacro
  ```

  Here, the macro uses the expression value of another macro as an argument to the Write command.

Macros can include comments, which are passed through as part of their definition.

### 9.1.1 Saving Macros

Macros can either appear in the file where they are invoked or, as is more common, in a separate include file. To place macros in an include file, in the Studio:

1. Select **Save** or **Save As** from the **File** menu.
2. In the **Save As** dialog, specify that the **Files of type** field has a value of Include File (*.inc).

   **Note:** The Macro Routine (*.mac) value for this field is not the correct file type for ObjectScript macros.
3. Enter the directory and file names, and save the macros.

9.2 Referring to Macros in External Files

When you have saved macros to a separate file, they can be included, with the \#Include directive:

\#Include MyMacros

Do not include the .inc suffix as part of the name of the referenced file.

When including macros within a class, the directives appear at the beginning of the class definition are of the form:

Include MyClassMacros
IncludeGenerator MyMethodGeneratorMacros

where the first line describes the standard syntax for including macros and the second line describes the syntax for including macros for use with method generators.

9.3 Calling Macros

When a macro's definition is part of a file, either by its definition or by using the \#include directive, you can then use it. To call a macro from within ObjectScript code, refer to it by its name prefixed with “$$$. Hence, if you have defined a macro called MyMacro, you can call it by referring to $$$.MyMacro.

Remember that macros are text substitutions. After a macro is substituted in, the syntax for the resulting statement is checked for correctness. Therefore, the macro defining an expression should be invoked in a context requiring an expression; the macro for a command and its argument can stand as an independent line of ObjectScript; and so on.

The code to invoke each of the macros is:

- For string substitutions, numeric substitutions, and variable substitutions:

  ```objectscript
  Write $$$.StringMacro
  Write $$$.NumericMacro
  Set Variable = 1 // this avoids the <UNDEFINED> error
  Write $$$.VariableMacro
  ```
For the command and argument-calling macro:

```objectscript
$$\text{CommandArgumentMacro}(\%\text{Run})
```

Since the macro refers to a complete line of ObjectScript syntax, the macro is on a line of its own. Note that the argument must begin with the “%” character, as it does when the macro is defined.

For the macro that uses a function, an expression, and an operator:

```objectscript
Write $$\text{FunctionExpressionOperatorMacro}
```

Here, the macro as a whole is an expression, so it must be treated as such. Other operations occur invisibly.

For the macro that refers to other macros:

```objectscript
$$\text{ReferenceOtherMacroMacro}
```

It is invoked on its own (as a complete line of code). Its definition and invocation can appear before or after the macro that it references.

### 9.4 Using Conditional Macros

To create optionally compiled code, use ObjectScript conditional macros. There are four of these:

- **#If** specifies the beginning of conditional content.
- **#ElseIf** specifies the end of conditional content defined by any preceding **#If** or **#ElseIf** statement and the beginning of another conditional case.
- **#Else** specifies the end of the last conditional content defined by any preceding **#If** or **#ElseIf** statement and the beginning of the default (sometime known as “fall-through” content).
- **#EndIf** specifies the end of a conditional content block.

An example of conditional macros is:
#If $Get(^Country)="England"
  Write "The capital is London."
#ElseIf $Get(^Country)="France"
  Write "The capital is Paris."
#Else
  Write "The capital Tallinn."
#EndIf
Multidimensional Arrays

Caché includes support for multidimensional arrays. A multidimensional array is a persistent variable consisting of one or more elements, each of which has a unique subscript identifier and where you can intermix different kinds of subscripts. An example is the following `MyVar` array:

- `MyVar`
- `MyVar(22)`
- `MyVar(-3)`
- `MyVar(“MyString”)`
- `MyVar(-123409, “MyString”)`
- `MyVar(“MyString”, 2398)`
- `MyVar(1.2, 3, 4, “Five”, “Six”, 7)`

The subscripts of `MyVar` are positive and negative real numbers, strings, and combinations of these. Additionally, each array node is a typical ObjectScript variable.

10.1 What Multidimensional Arrays Are

Succinctly, multidimensional arrays are persistent, n-dimensional arrays that are denoted through the use of subscripts. Individual nodes are also known as “globals” and are the building block of Caché data storage. They have other characteristics as well:

- They exist in tree structures.
• They are sparse.
• They can appear in multiple settings.

### 10.1.1 Multidimensional Tree Structures

The entire structure of a multidimensional array is called a *tree*; it begins at the top and grows downwards. The *root*, `MyVar` above, is at the top. The root, and any other subscripted form of it, are called *nodes*. Nodes that have no nodes beneath them are called *leaves*. Nodes that have nodes beneath them are called *parents* or *ancestors*. Nodes that have parents are called *children* or *descendants*. Children with the same parents are called *siblings*. All siblings are automatically sorted numerically or alphabetically as they are added to the tree.

### 10.1.2 Sparse Multidimensional Storage

Multidimensional arrays are sparse. This means that the example above uses only seven reserved memory locations, one for each defined node. Further, since there is no need to declare arrays or specify their dimensions, there are additional memory benefits: no space is reserved for them ahead of time; they use no space until needing it; and all the space that they use is dynamically allocated. As an example, consider an array used to keep track of players' pieces for a game of checkers; a checkerboard is 8 by 8. In a language that required an 8–by-8 checkerboard-sized array would use 64 memory locations, even though no more than 24 positions are ever occupied by checkers; in ObjectScript, the array would require 24 positions only at the beginning, and would need fewer and fewer during the course of the game.

### 10.1.3 Settings for Multidimensional Arrays

Multidimensional arrays can appear in three different settings:

- Any global can be used and thereby transformed into an array. In this case, global `^y` becomes a node in the array `y` when you create global `y(1)`.
- Any property can be used and thereby transformed into an array. In this case, property `Object.Get(Prop)` becomes a node in the array `Person.Get(Prop)` when you create property `Person.Get(Prop,1)`.
- Any local variable can be used and thereby transformed into an array. In this case, variable `x` becomes a node in array `x` when you create variable `x(1)`. 
10.2 Manipulating Multidimensional Arrays

You can write to and read from them using the Read and Write commands respectively. They are easily identified by their leading “^” (caret) character.

Caché provides a comprehensive set of commands and functions for working with multidimensional arrays:

- **Set** places values in them.
- **Kill** removes all or part an array structure.
- **$Order** and **$Query** allows you to iterate over the contents of an array.
- **$Data** allows you to test for the existence of nodes in an array.

This set of commands and functions can operate on multidimensional globals and multidimensional local variables.

10.3 For More Information

For further information on multidimensional arrays, see Using Caché Multidimensional Storage.
ObjectScript provides several groups of operations related to strings, each with its own purpose and features. These are:

- **Basic String Operations and Functions**
- **Delimited String Operations**
- **List-Structure String Operations**

### 11.1 Basic String Operations and Functions

ObjectScript basic string operations allow you to perform various manipulations on a string. They include:

- **The $Length function** returns the number of characters in a string: For example, the code:
  ```objectscript
  Write $length("How long is this?")
  ```
  returns 17, the length of a string.

- **$Justify** returns a right-justified string, padded on the left with spaces (and can also perform operations on numeric values). For example, the code:
  ```objectscript
  Write "one",!,$justify("two",8),!,"three"
  ```
  justifies string “two” within eight characters and returns:
  ```text
  one
  two
  three
  ```
$ZConvert converts a string from one form to another. It supports both case translations (to upper case, to lower case, or to title case) and encoding translation (between various character encoding styles). For example, the code:

Write $ZConvert("cRAZy cAPs","t")

returns:
CRAZY CAPS

The $Find function searches for a substring of a string, and returns the position of the character following the substring. For example, the code:

Write $find("Once upon a time...", "upon")

returns 10 character position immediately following “upon.”

The $Extract function, which returns a substring from a specified position in a string. For example, the code:

Write $Extract("Nevermore"),$Extract("prediction",5),$Extract("xon/xoff",1,3)

returns three strings. The one-argument form returns the first character of the string; the two-argument form returns the specified character from the string; and the three-argument form returns the substring beginning and ending with specified characters, inclusive. In the example above, there are no line breaks, so the return value is:

Nixon

11.1.1 Advanced Features of $Extract

You can use the $Extract function in conjunction with the Set command pad a string on the left with spaces.

Set x = "abc"
Write x,
Set $Extract(y, 3) = x
Set x = y
Write x

This code takes the string “abc” and places at the third character of string y. Because y has no specified value, $Extract assumes that its characters are blank, which acts to pad the string.

You can also use $Extract to insert a new string at a particular point in variable. It extracts the characters specified and replaces them with the supplied substring, whether or not the lengths of the old and new strings match. For example:
Set x = "1234"
Write x,!
Set $Extract(x, 3) = "abc"
Write x,!
Set $Extract(y, 3) = "abc"
Write y

This code sets $x to “1234” ; it then extracts the third character of $x using $Extract and inserts “abc” in its place, making the string “12abc4”.

11.2 Delimited Strings

Caché includes functionality that allows you to work with strings as a set of substrings. This functionality provides for the manipulation of related pieces of data that you wish to store as a single whole. These are

- **$Piece**—Returns a specific piece of a string based on a specified delimiter. It can also return a range of pieces, as well as multiple pieces from a single string, based on multiple delimiters.

- **$Length**—Returns the number of pieces in a string based on a specified delimiter.

The $Piece function provides uniquely important functionality because it allows you to use a single string that contains multiple substrings, with a special delimiter character (such as “^”) to separate them. The large string acts as a record, and the substrings are its fields.

The syntax for $Piece is:

Write $Piece("ListString","QuotedDelimiter",ItemNumber)

where ListString is a quoted string that contains the full record being used; QuotedDelimiter is the specified delimited, which must appear in quotes; and ItemNumber is the specified substring to be returned. For example, to display the second item in the following space-delimited list, the syntax is:

Write $Piece("Kennedy Johnson Nixon"," ",2)

which returns “Johnson”.

You can also return multiple members of the list, so that the following:

Write $Piece("Nixon***Ford***Carter***Reagan","***",1,3)
returns “Nixon***Ford***Carter”. Note that both values must refer to actual substrings and
the third argument (here 1) must be a smaller value than that of the fourth argument (here
3).

The delimiter can be anything you choose, such as with the following list:

```
Set x = $Piece("Reagan,Bush,Clinton,Bush","",3)
Set y = $Piece("Reagan,Bush,Clinton,Bush","Bush",2)
Write x,!,,y
```

which returns

Clinton
,Clinton,

In the first case, the delimiter is the comma; in the second, it is the string “Bush”, which is
why the returned string includes the commas. To avoid any possible ambiguities related to
delimiters, use the List-related functions, described in the next section.

### 11.2.1 Advanced $Piece Features

A call to $Piece that sets the value of a delimited element in a list will add enough list items
so that it can place the substring as the proper item in an otherwise empty list. For instance,
if you set the fourth item in a list,

```
Set $Piece(Alphalist, "^", 1) = "a"
Set $Piece(Alphalist, "^", 20) = "t"
Write $Length(Alphalist,"^")
```

will return a value of 20, since it creates twenty delimited items. However, items 2 through
19, in this example, will not have values set. Hence, if you attempt to display any of their
values, nothing will appear.

A delimited string item can also contain a delimited string. To retrieve a value from a sublist
such as this, nest $Piece function calls, as in the following code:

```
Set $Piece(Powers, "^", 1) = "1::1::1::1::1"
Set $Piece(Powers, "^", 2) = "2::4::8::16::32"
Set $Piece(Powers, "^", 3) = "3::9::27::81::243"
Write Powers,!
Write $Piece( $Piece(Powers, "^", 2), "::", 3)
```

This code returns two lines of output: the first is the string Powers, including all its delimiters;
the second is 8, which is the value of the third element in the sublist contained by the second
element in Powers. (In the Powers list, the nth item is a sublist of two raised to the first
through fifth powers, so that the first item in the sublist is n to the first power, and so on.)
11.3 List-Structure String Operations

ObjectScript defines a special kind of string called a “list”, which consists of a list of substrings. There are a set of functions for manipulating lists, which are:

- **$ListBuild**—Returns a list built from substrings.
- **$List**—Returns a specific item in the list.
- **$ListLength**—Returns the number of items in the list.
- **$ListFind**—Searches the list for a substring and returns its item number.

Though consisting of standard strings, Caché treats lists slightly differently than standard strings. Because of this, you should not use standard string functions on lists. Further, using a list function on a regular string generates a <LIST> error.

The following procedure demonstrates the use of the various list functions:

```objectscript
ListTest() PUBLIC {
    // set values for list elements
    Set Addr = "One Memorial Drive"
    Set City = "Cambridge"
    Set State = "MA"
    Set Zip = "02142"

    // create list
    Set Mail = $ListBuild(Addr, City, State, Zip)

    // get user input
    Read "Enter a string: ", input, !, !

    // if user input is part of the list, print the list's content
    if $ListFind(Mail, input) {
        for i=1:1:$ListLength(Mail) {
            Write $List(Mail, i), !
        }
    }
}
```

This procedure demonstrates several notable aspects of lists:

- **$ListFind** only returns true if the value being tested matches the list item exactly.
- **$ListFind** and **$ListLength** are used in expressions.
11.3.1 Advanced List Features

A call that sets the value of an element in a list will add enough list items so that it can place the substring as the proper item in an otherwise empty list. For instance, if you set the fourth item in a list,

```
Set $List(Alphalist, 1) = "a"
Set $List(Alphalist, 20) = "t"
Write $Listlength(Alphalist)
```

will return a value of 20, since it creates twenty items for the list. However, items 2 through 19, in this example, will not have values set. Hence, if you attempt to display any of their values, you will receive a <NULL VALUE> error.

A list item can also be a list. To retrieve a value from a sublist such as this, nest $List function calls, as in the following code:

```
Set $List(Powers, 2)=$ListBuild(2,4,8,16,32)
Write $List( $List(Powers, 2), 5)
```

This code returns 32, which is the value of the fifth element in the sublist contained by the second element in the Powers list. (In the Powers list, the second item is a sublist of two raised to the first through fifth powers, so that the first item in the sublist is two to the first power, and so on.)

11.4 Lists Versus $Piece and Strings

Though the $Piece function allows you to manage lists, it depends on delimiters; a list is useful for avoiding delimiters altogether. With delimiters, there's always the chance that one of the substrings will contain the delimiter character(s), which will throw off the positions of the pieces in the list. Lists prevent this.

One possible benefit of the $Piece-oriented approach is that it allows you to more flexibly search its content, using the $Find function. Because $ListFind requires an exact match, you cannot search for random substrings in lists. Hence, in the example above, searching on the string “One” will return Zero, indicating failure, even though the address begins with the characters “One”. At the same time, this potential benefit must be weighed against the ease of use of the list features as a whole.
12

Calling Out of Caché

This chapter describes utilities and functions that enable you to access external commands from within Caché. Topics include:

- Issuing Operating System Commands from Caché
- Calling External Applications with $ZF
- Building Callout Modules as Shared Libraries
- Referencing a User-Defined Callout Module
- Sample Caché Callout DLL Build Procedures
- Reading and Writing to Devices Opened in UNIX

12.1 Issuing Operating System Commands from Caché

Caché allows you to issue operating systems commands on supported platforms as follows:

- OpenVMS platform to issue OpenVMS DCL commands
- UNIX platforms to issue UNIX shell commands
- Windows platforms to issue MS-DOS commands

There are multiple methods for issuing such commands:
12.1.1 Issuing Operating System Commands with %CLI

The %CLI utility is a tool for issuing operating system commands (DCL, UNIX shell, Windows) from the Caché command-line prompt (for example, from the Caché Terminal). Although similar to the function $ZF(-1,command), %CLI only works from the Caché command line prompt and cannot be used in routines.

On Windows, use %CLI to create a test directory as follows:

%SYS> Do ^%CLI
CLI command: mkdir test

Note: On Windows, use %CLI only with commands that have their effects in the background. If a command asks the user for input, such as the Time command, the process may hang.

On UNIX, use %CLI to call the UNIX shell command pwd as follows:

USER> Do ^%CLI
CLI command: pwd
/usr/cachesys

On OpenVMS, use %CLI to call the DCL command SHOW DEFAULT (show the current namespace):

USER> Do ^%CLI
CLI command: SHOW DEF
DUA0:[usr]

12.1.2 Issuing Operating System Commands from Caché with $ZF(-1)

Caché provides a special $ZF(-1) function for:

• Issuing commands to Windows.
• Issuing shell commands to UNIX.
• Calling UNIX or OpenVMS system services.
• Issuing DCL commands to OpenVMS.
12.1.2.1 Issuing UNIX Shell and Windows Commands

When you use $ZF to call a UNIX shell or Windows command, Caché calls the UNIX shell or the Windows command processor, which issues the UNIX call or Windows command and places the return code from that call in $ZF.

Invoke $ZF to issue a shell or operating system command as follows:

```
Set CallResults = $ZF(-1,command)
```

- **CallResults**—exit code from the specified command.
- **command**—shell or operating system command line you want to execute.

Use $ZF(-1,...) only with Windows commands that have their effect in the background, as in the example above. If a command asks the user for input, such as the time command, the process may hang.

Use $ZF(-1) without a second argument to load an interactive subshell. The default shell is “sh”, but you can override this by setting the shell environment variable SHELL to an alternate shell.

The following UNIX command example displays the output of a shell pwd command. The return status will be in status.

```
USER> Set status = $ZF(-1,"pwd")
/usr/cachesys
USER>
```

The following Windows command creates a test directory. The return status will be in status.

```
USER> Set status = $ZF(-1,"mkdir test")
```

**Equivalence of $ZF(-1) and “!”**

You can also use the “!” character (exclamation point) to load an interactive subshell. If you begin a line in Caché programmer mode with “!” , the rest of the line is executed in a subshell. All leading white space (either blank spaces or tabs) is stripped.

In other words, the following uses of “!” at the Caché prompt are equivalent to the indicated uses of $ZF:

<table>
<thead>
<tr>
<th>Command</th>
<th>is equivalent to</th>
<th>$ZF(-1,command)</th>
</tr>
</thead>
<tbody>
<tr>
<td>!command</td>
<td>is equivalent to</td>
<td>$ZF(-1)</td>
</tr>
</tbody>
</table>

```
For example, if you enter the following command in UNIX, the terminal will display the output of a shell `pwd` command.

```
USER> !pwd
/usr/cachesys
USER>
```

### 12.1.2.2 Issuing OpenVMS DCL Commands

When you use `$ZF` to call an OpenVMS DCL command, Caché spawns a process that issues the DCL command and places the return code from that call in `$ZF`. Whatever the privilege level required for the DCL call, the process making the call returns to its original privilege level after the command is completed. If the OpenVMS process created when you issue `$ZF(-1)` is killed or stopped, the return code is -1.

Invoke `$ZF` to issue a DCL command as follows:

```
Set CallResults = $ZF(-1,DCLcommand[,output_device[,input_device]])
```

where

<table>
<thead>
<tr>
<th>CallResults</th>
<th>Value returned when the DCL subprocess exits</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCLcommand</td>
<td>DCL command line to execute</td>
</tr>
<tr>
<td>output_device</td>
<td>Definition of SYS$OUTPUT for the subprocess (optional). Uses current SYS$OUTPUT if not specified.</td>
</tr>
<tr>
<td>input_device</td>
<td>Definition of SYS$INPUT for the subprocess (optional). Uses current SYS$INPUT if not specified.</td>
</tr>
</tbody>
</table>

For example, this command displays on the terminal the output of a DCL `SHOW DEF` command. The return status will be in `CallResults`.

```
USER> Set CallResults = $ZF(-1,"SHOW DEF")
DUA0:[usr]
```

### 12.1.3 Issuing Operating System Commands from Caché with `$ZF(-2)`

The operation of `$ZF(-2)` is essentially the same as `$ZF(-1)`, except that Caché launches the specified command in what is called “detached” (or asynchronous) mode. It does not wait for the process running the system command to complete.

Commands initiated via `$ZF(-2)` should not perform I/O on the principal device. Typically, Caché equates the principal device to the null device.
12.2 Calling External Applications with $ZF

Using $ZF, you can call routines written in languages other than Caché ObjectScript:

- In Windows, you can call routines written in protocols compatible with C and Windows commands.
- In UNIX, you can call routines written in protocols compatible with C and UNIX system services.
- In OpenVMS, you can call routines written in other languages including C, macro assembler, OpenVMS system services, and OpenVMS library routines. For specific function calls for DSM compatibility and conversion, refer to $ZF Function Calls for DSM.

$ZF has two startup parameters, **Maximum String Size** and **Maximum Heap Size**. **String Size** can range from 2 to 32,767 bytes (with a default of 32,767 bytes) and **Heap Size** can range from 0 to 67,584 bytes (with a default of 67,584 bytes). To modify these, see the Process section of the Advanced tab of the Caché Configuration Manager.

12.2.1 Calling External Applications with $ZF on OpenVMS Platforms

To call external applications from OpenVMS, create a file, called CZF.EXE. This file contains the information about the routines needed by Caché. The sections that follow explain how to create CZF.EXE.

InterSystems recommends that you first develop CZF.EXE in a private directory. You can then move it to the configuration's bin subdirectory after testing it.

If you requested the link libraries (that is, answered “Yes” to the installation prompt “Do you want to load the Caché Engine link libraries?”), then all needed support files will be found in <install-dir>:[SOURCE.CACHE], where <install-dir> is the root directory for your configuration. You can use this directory to build CZF.EXE, or copy the files to another location if desired. They can also be extracted from the distribution media.

1. Copy the Caché Engine link libraries to your private build directory, if necessary.
2. Design and code the external routines.
3. Compile or assemble the external routines to create .obj or .mac files.
4. Describe the routines to be called in one of the two supplied CZF files.

5. If you used CZF.C, compile it. If you used CZF.M64, assemble it.

6. Use CZFBUILD.COM to link the object files to create CZF.EXE.

7. Define a logical CZF that points to the working directory.

8. Sign on to Caché.

9. Issue $ZF calls from Caché to access the external routines.

10. Check that the routine succeeded.
    
    When your testing has been successful and you are ready to incorporate your custom CZF.EXE into the configuration for all your users, continue with the following, concluding steps.

11. Remove the logical for CZF.

12. Shut Caché down.

13. Install your new CZF.EXE.


**12.2.1.1 Step 1: Go to Working Directory and Copy CZF Files**

Make the current and default namespace the private directory in which you will build CZF.EXE. To do this, issue the following commands at the DCL prompt:

```bash
$ SET DEF work_dir
```

Then, if necessary, copy in the CZF files from the [SOURCE.CACHESYS] directory (or whatever the installation directory may be): CZF.C, CZF.M64, CZFBUILD.COM, and CDZF.H.

**12.2.1.2 Step 2: Design and Code External Routines**

Caché calls the routine with a protocol that is compatible with C. When you install the routine, you describe the input and output arguments. The description indicates the data types of the arguments, how they are passed, and which are output arguments. The value the routine assigns to the output argument is the value returned by $ZF to the Caché program.

The return code of the routine indicates whether the routine succeeded. Return a 1 (one) if the routine was successful; return an error code if it was not. If it returns an even error code, $ZF produces a <FUNCTION> error.
This example assumes that you describe the first argument of routine ABC as an output argument. Now assume that a Caché routine calls ABC like this:

```
Set VAL = $ZF("ABC",X)
```

The calling routine passes to the routine ABC a copy of the current value of variable X. ABC can replace that value with a new one. This action does not affect the value of X. The example assigns the value that ABC returns to VAL; X remains unaffected.

Caché stores fractional numbers in base ten form. For example, the number 1.3 is stored as 13 times one tenth. In contrast, most high-level languages store fractional numbers in base two form: the number 1.3 is translated into a binary number times some power of two. If you describe an argument as floating point, Caché converts its value from base ten to base two on input and back to base ten on output. This conversion may introduce slight inaccuracies.

### 12.2.1.3 Step 3: Compile or Assemble the External Routines

Compile or assemble the routines, depending on the language in which they are written.

### 12.2.1.4 Step 4: Describe the External Routines in a CZF File

Caché provides two files to describe external routines accessed by a $ZF function call:

- CZF.C
- CZF.M64

Use CZF.C if you have a C compiler; otherwise, use CZF.M64, written in macro.

#### Using CZF.C

The CZF.C file has the following basic form.

```c
#include "cdzf.h" /* Always include this line*/
#include <stdio.h> /* printf, etc. */
#include <string.h> /* string and strcopy definitions*/
#include <descrip.h> /* VMS Descriptor definitions */
#include <starlet.h> /* VMS system service definitions */
#include <lib$routines.h> /* VMS library routines */

ZFBEGIN

/* You need one ZFENTRY statement for each entry point */
/* in a external routine you will call, in the form shown below. */

 ZFENTRY("name","argtypes",entry-point)
  ... 
ZFEND

/* If you prefer, you can put C routines here; however, */
/* InterSystems recommends they exist in separate files */
/* for maintenance. */
```

Using Caché ObjectScript
ZFENTRY Statements That Maintain User Privilege Level

Include a ZFENTRY statement in CZF.C for each external routine that you will run at the privilege level of the calling process. Locate these statements between the ZFBEGIN and ZFEND statements for the following:

- Each entry point you will call in the external routines
- Each OpenVMS library routine
- Each OpenVMS system service call

A ZFENTRY statement has the following form:

```
ZFENTRY("name","argtypes","entrypoint")
```

where:
<table>
<thead>
<tr>
<th><strong>name</strong></th>
<th>The name by which Caché calls the external routine in a $ZF function call. This can, but does not have to, match the entrypoint parameter name.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>argtypes</strong></td>
<td>A string that describes how the arguments are passed. It can contain zero or more of the following characters, in a quoted form &quot;&quot;&quot;:</td>
</tr>
<tr>
<td></td>
<td>• i—an integer</td>
</tr>
<tr>
<td></td>
<td>• p—a pointer to an integer</td>
</tr>
<tr>
<td></td>
<td>• c—a pointer to a zero-terminated character string (8-bit characters)</td>
</tr>
<tr>
<td></td>
<td>• f—a pointer to a floating-point number</td>
</tr>
<tr>
<td></td>
<td>• d—a pointer to a double-precision number</td>
</tr>
<tr>
<td></td>
<td>• v—a pointer to an OpenVMS fixed length string descriptor</td>
</tr>
<tr>
<td></td>
<td>• b—a pointer to a counted byte array (see the reference page for $ZF for information on zarray)</td>
</tr>
<tr>
<td></td>
<td>• t—a pointer to a translated string (in counted byte array)</td>
</tr>
<tr>
<td></td>
<td>• s—a pointer to a counted wide string (16-bit characters; see the reference page for $ZF for information on zwarray)</td>
</tr>
<tr>
<td></td>
<td>• w—a pointer to a zero-terminated wide character string (16–bit characters)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For example, “ip” indicates an integer and a pointer to an integer. If you use an uppercase “P,” “C,” “F,” “D,” or “V,” you indicate that the argument is also an output argument. This means the routine will return a value in that Caché variable. If you specify more than one output argument, $ZF returns the arguments as a string delimited by commas.</td>
</tr>
<tr>
<td></td>
<td>Structure and argument prototype definitions can be seen in the include file cdzf.h. Additional InterSystems internal definitions are also declared there.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Due to normal C calling conventions, arguments that are passed as integers cannot be output arguments.</td>
</tr>
<tr>
<td><strong>entry-point</strong></td>
<td>The actual entry point name in the external routine. There must be a matching extern statement for each entry point.</td>
</tr>
</tbody>
</table>
ZFPRIV Statements To Run at Elevated Privilege Level

Describe the external routine with a ZFPRIV statement if you want it to have all privileges except the SHARE privilege. A ZFPRIV statement has the same format as a ZFENTRY statement except for the statement name and the addition of a privileges argument at the end of the argument list, as shown below:

ZFPRIV("name","argtypes","entrypoint","privileges")

The privileges argument can be either of the following:

- PRIV—routine runs at elevated privilege level
- NOPRIV—routine will run at privilege level of calling process

CZF.C Example

This C routine translates strings to uppercase.

```c
mupper(string)
    char *string;
    {
        while (*string)
        { if (*string >= 'a' && *string <= 'z')
            *string += 'A' - 'a';
            ++string;
        }
        return(0);
    }
```

This C routine is in the file upper.c. The following ZFENTRY statement describes the C routine:

ZFENTRY("UPPER","c",mupper)

Caché calls this C routine as follows:

Set LOW="abcdef"
Set HIGH=$ZF("UPPER",LOW)
Write HIGH

This code returns "ABCDEF"; after executing, LOW still contains “abcdef” and HIGH contains “ABCDEF”.

A second C routine, in the file sum3.c, returns the sum of three numbers.

```c
sum3(a,b,c,d)
    float *a,*b,*c;
    double *d;
    { 
        *d = *a + *b + *c;
        return(0);
    }
```
The following `ZFENTRY` statement describes this C routine:

```
ZFENTRY("ADDTHREE","fffD",sum3)
```

Caché calls this C routine as follows:

```
Set SUM = $ZF("ADDTHREE",50,42.76,V)
```

The `CZF.C` file for these routines is:

```
#include "cdzf.h"
#include "upper.c"
#include "sum3.c"

extern int mupper();
extern int sum3();

ZFBEGIN
ZFENTRY("UPPER","C",mupper)
ZFENTRY("ADDTHREE","fffD",sum3)
ZFEND
```

For other examples, look at the file `CZF.C` supplied with Caché.

**Using CZF.M64**

If you use the `CZF.M64` file, you enter only one type of statement between the `.SHOW` and `.END` commands. The statement is called `ZFENTRY` and has the following format:

```
ZFENTRY CALLNAME=name,LINKNAME=fname,ARGS=atype,PRIVS=priv
```

- **CALLNAME**—The name by which Caché calls the external routine in a $ZF function call.
- **LINKNAME**—The entry point name of the user-written external routine. Equivalent to the entrypoint argument in the `ZFENTRY` statement used with `CZF.C`.
- **ARGS**—A string that describes how the arguments are passed. It can contain zero or more of the following characters, placed between quotation marks:
  - `i`—an integer
  - `p`—a pointer to an integer
  - `c`—a pointer to a character string
  - `f`—a pointer to a floating-point number
  - `d`—a pointer to a double precision number
  - `v`—a pointer to an OpenVMS fixed-length string descriptor
- b—a pointer to a counted byte array (see the reference page for $ZF for information on zarray)
- t—a pointer to a translated string (in counted byte array)
- s—a pointer to a counted wide string (16-bit characters; see the reference page for $ZF for information on zwarray))
- w—a pointer to a zero-terminated wide character string (16–bit characters)

For example, "ip" indicates an integer and a pointer to an integer. If you use an uppercase P, C, F, D, or V, you indicate that the argument is also an output argument. This means the routine will return a value in that Caché variable. If you specify more than one output argument, $ZF returns the arguments as a string delimited by commas.

**Note:** Due to normal C calling conventions, arguments that are passed as integers cannot be output arguments.

Structure and argument prototype definitions can be seen in the include file cdzf.h. Additional InterSystems internal definitions are also declared there.

• **PRIVS**—Enter “YES” to have the routine run at an elevated privilege level. Enter “NO” to have the routine run at the same privilege level as the calling routine.

### 12.2.1.5 Step 5: Compile or Assemble the CZF File

To prepare a CZF.OBJ file that you link with the external routines and Caché, you need to compile or assemble the CZF source file.

**Compile CZF.C**

To compile CZF.C, issue the following command from the DCL prompt:

```
$ CC CZF.C
```

**Assemble CZF.M64**

To assemble CZF.M64, issue the following command from the DCL prompt:

```
$ MAC /ALPHA CZF.M64
```

### 12.2.1.6 Step 6: Use CZFBUILD.com to Create CZF.EXE

Use the CZFBUILD.com file to link in the external routines. You edit CZFBUILD.com to add the files you will link. CZFBUILD.com looks like this before you edit it:
1. Edit CZFBUILD.COM to include CZF.OBJ and each of the external object files. Add each filename containing external routines for which you defined in the CZF file to the second line in CZFBUILD.COM. For instance, if the routines are in two files named abc.obj and xyz.obj, the second line will look like this after you edit it:

   CLUSTER=CZF,,,CZF,ABC,XYZ

2. Run CZFBUILD.COM at the DCL prompt. This creates the file, CZF.EXE.

12.2.1.7 Step 7: Define a Logical for CZF

You need to let Caché know where to find the CZF.EXE file you want to test. To do this, define a logical by entering the following command at the DCL prompt:

   $ DEFINE CZF DEVICE: [YOURDIR]CZF.EXE

12.2.1.8 Step 8: Sign on to Caché

Sign on to Caché so you can use the external routines:

   CSESSION <configname>

where <configname> is the installation of Caché that you want to use.

12.2.1.9 Step 9: Issue $ZF Calls from Caché

To call a routine you defined in the CZF file, use this form of $ZF:

   Set CallResults = $ZF(RoutineName,input1,...,inputn,output1,...,outputn)

- CallResults — Holds the result of the routine call as a string. The pieces are output parameter values (if they exist), separated by commas within the string. You retrieve each output by using the $PIECE command.

- RoutineName — The name of the routine as defined in the CALLNAME variable of the ZFENTRY in the CZF.M64 file or the Name variable of the ZFENTRY in the CZF.C file

(For specific function calls for DSM compatibility and conversion, refer to $ZF Function Calls for DSM.)
For instance, if you made a \texttt{$ZF$} call with three output parameters and you want to see the value of the second one, do the following:

\begin{verbatim}
Set CallResults = $ZF("USER_PROG",in1,out1,out2,out3)
WRITE $PIECE(CallResults","",2)
\end{verbatim}

The value returned in \texttt{out2} is displayed. The example assumes you have preset the size of the output parameters to be the maximum you would expect from the external routine you called.

Another example is this \texttt{$ZF$} call, which passes the value of the Caché variable \textit{ANGLE} to the user-written routine \textit{SIN}. It assigns to \texttt{X} the value returned for \texttt{$ZF$}:

\begin{verbatim}
Set X = $ZF("SIN",ANGLE)
\end{verbatim}

### 12.2.1.10 Step 10: Verify that the Routine Succeeded

Does the routine perform as you expect? Does it return an error?

A \texttt{<FUNCTION>} error from \texttt{$ZF$} means one of two things:

- The routine you want to call has not been installed.
- The routine itself returned an even number, which is interpreted as an error code.

#### Errors from OpenVMS System Service Calls

Most OpenVMS System Service routines return a condition value in Register R0. The return value contains the completion status of the operation in bits three to zero of R0. Caché uses the contents of R0 to determine whether or not the \texttt{$ZF$} routine call completed normally. If the value returned is even, Caché issues a \texttt{<FUNCTION>} error. You can examine this error using the \texttt{SYSLOG} utility.

For specific VMS function calls for DSM compatibility and conversion, refer to \texttt{$ZF$} Function Calls for DSM.

### 12.2.1.11 Step 11: Remove the logical for CZF

Issue this command at the DCL prompt:

\begin{verbatim}
$ DEASSIGN CZF
\end{verbatim}

### 12.2.1.12 Step 12: Shut Down Caché

Issue the following command:

\begin{verbatim}
CCONTROL STOP <configname>
\end{verbatim}

where \texttt{<configname>} is the installation of Caché to shut down.
12.2.1.13 Step 13: Install New CZF.EXE

You might need to unload the Caché executables from shared memory in order to replace CZF.EXE. Use:

```
CCONTROL UNLOAD <configname>
```

where `<configname>` is the installation of Caché to unload.

Then, copy CZF.EXE from your private directory to the `<install-dir>:[bin]` subdirectory, for example:

```
COPY CZF.EXE DKA0:[CACHESYS.BIN]
```

12.2.1.14 Step 13: Restart Caché

To restart Caché, issue the following command:

```
CCONTROL START <configname>
```

where `<configname>` is the installation of Caché to start.

Privileges Controlled For Security

There can be security problems if the following three conditions are true:

- The logical CZF is defined.
- That logical is defined outside of the system logical name table.
- That logical is not defined in executive mode.

To avoid security problems, privileges are dropped before calling a `$ZF` routine regardless of the setting of the `PRIVS` argument in the `ROUTINE` macro if the above three conditions are all true.

If you need to test functions with privileges, you must define the logical name CZF in executive mode, as in:

```
DEFINE/EXEC CZF DUA0:[USER29]CZF.EXE
```

You must have OpenVMS SYSNAM privilege to successfully issue this command.

Saving Changes When you Upgrade Caché

When you install a new version of Caché, the installation process checks to see if you have existing CZF.C, CZF.M64, and CZF.EXE files. If you have a CZF.EXE, it will be renamed and the one from the kit used, to ensure compatibility with the new version. You can replace
or re-link the prior one. Any source files (such as CZF.C or CZF.M64) will be retained and
the ones from the kit will be copied in as prototypes for possible upgrading.

**12.2.2 Calling External Functions with $ZF on UNIX or Windows Platforms**

The primary source for external functions is the C programming language. Use $ZF as follows
to call a user-written function from Caché:

```objectscript
Set var=$ZF("function-name",args)
```

- **var** — a variable that receives the return value.
- **function-name** — the name of the function being called.
- **args** — Optional argument values passed to the function, in the form: `arg1, arg2, arg3, ...
  ...argn`

For example, the following command passes the value of the Caché variable `ANGLE` to the
user-written function `SIN`. It assigns to `X` the value returned by $ZF.

```objectscript
Set X = $ZF("SIN",ANGLE)
```

**12.2.2.1 Development Steps**

The following table gives an overview of the steps to design and develop external functions.
Each step is described in detail in the sections following the table.

1. Create a private directory for the external functions.
2. Design and code the external functions.
3. Create a file of external functions and descriptions.
4. Compile or assemble the external functions.
5. Link the object file and Caché object files into a new version of Caché.
7. Replace the Caché executable with the new file from step 5.
8. Restart Caché.
12.2.2.2 Step 1: Create a Private Directory

If you requested the link libraries (that is, answered “Yes” to the UNIX installation prompt of “Do you want to load the Caché Engine link libraries?” or checked the equivalent box for a Windows installation), all needed support files will be found in `\<install-dir>\source\Cache`, where `<install-dir>` represents the path to the installation root directory for your configuration. You can use this directory to build the new Caché executable, or copy the files to another location, if desired. They can also be extracted from the installation media.

12.2.2.3 Step 2: Design and Code External Routines

Caché calls your functions with a protocol that is compatible with C. When you install the functions, you describe the input and output arguments. The description indicates:

- The data types of the arguments
- How the arguments are passed
- Which arguments are output arguments

The value the function assigns to the output argument is the value returned by `$ZF` to the Caché program.

The return code of the function indicates whether the function succeeded. If the return code is 0 (zero), the function call was successful. If the return code is non-zero, `$ZF` reports a `<FUNCTION>` error.

If you get a `<FUNCTION>` error from `$ZF`, it means either:

- The function you called was not installed.
- The function returned an error code.

This example assumes that you describe the first argument of the function ABC as an output argument. Assume that a Caché routine calls ABC as follows:

```ObjectScript
Set VAL = $ZF("ABC",X)
```

The calling Caché routine passes a copy of the current value of X to ABC. ABC can then replace the passed value of X with a new value. This action does not affect the value of X since the Caché routine assigns the value of X that ABC returns to VAL. X remains unaltered.

Caché stores fractional numbers in base 10 form. For example, the number 1.3 is stored as 13*.10.
In contrast, most high-level languages store fractional numbers in base 2 form. For example, 1.3 is translated into a binary number times some power of two. If you describe an argument as floating point, Caché converts its value from base 10 to base 2 on input and back to base 10 on output. This conversion may introduce slight inaccuracies.

**Additional Call-Out Signal Processing (UNIX Only)**

Some system calls may fail if the process receives a signal, the most common being: **open, read, write, close, ioctl,** and **pause.**

If the function uses any of these system calls, you have to code carefully to be able to distinguish among real errors, a **Ctrl-C,** and a call that should be restarted. A small set of functions was created to allow you to check for asynchronous events and to set a new alarm handler in **$ZF.**

The function declarations are included in **cdzf.h:**

- **int sigrtclr();** — Clears retry flag. Should be called once before using **sigrtchk().**
- **int dzfalarm();** — Establishes new SIGALRM handler.

  On entry to **$ZF,** the previous handler is automatically saved. On exit, it is restored automatically. A user program should not alter the handling of any other signal.

- **int sigrtchk();** — Checks for asynchronous events. Should be called whenever one of the following system calls fails: **open, close, read, write, ioctl, pause,** or any call that fails when the process receives a signal. It returns a code indicating the action that the user should take:
  - -1 — Not a signal. Check for I/O error. See contents of errno variable.
  - 0 — Other signal. Restart operation from point at which it was interrupted.
  - 1 — SIGINT/SIGTERM. Exit from **$ZF** with a SIGTERM "return 0". Caché traps these signals appropriately.

In a typical **$ZF** function used to control some device, you would code something like this:

```objectscript
if ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
    Set some flags
    Call zferror
    return 0;
}
```

The **open** system call may fail if the process receives a signal. Usually this situation is not an error and the operation should be restarted. Depending on the signal, however, you might take other actions. So, in order to take account of all the possibilities, consider using the following code:
sigrtclr();
while (TRUE) {
  if (sigrtchk() == 1) return 1 or 0;
  if ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
    switch (sigrtchk()) {
    case -1:
      /* This is probably a real device error */
      Set some flags
      Call zferror
      return 0;
    case 0:
      /* A innocuous signal was received. Restart. */
      continue;
    case 1:
      /* Someone is trying to terminate the job. */
      Do cleanup work
      return 1 or 0;
    }
  } else break;
}

Code to handle the normal situation:
open() system call succeeded

Remember you must not set any signal handler except via dzfalarm.

12.2.2.4 Step 3: Create C Function Descriptions

Create a file called uzf.c to describe the C functions you want to incorporate into Caché. The uzf.c file should have the following basic form:

```c
#include "cdzf.h" /* Always include this line */
extern int entry-point ( /* Use one extern statement per function */
  #include filename /* Use one include statement for each file of C functions */
  ... /* Put additional extern statements here */
  ZFBEGIN
  ZFENTRY("name","argtypes",entrypoint)
  ... /* You need one ZFENTRY statement for each C function */
  ZFEND
  ... /* C functions go here if they are not included */
```

You must use the extern statement to declare all the user-written C functions before the ZFBEGIN statement. Write one ZFENTRY statement for each C function between the ZFBEGIN and the ZFEND.

You can place the functions directly in the uzf.c file after the ZFEND statement or you can place the functions in separate files and put an include statement in uzf.c for each of them. This latter solution produces cleaner code.

The parameters of ZFENTRY and the symbols you use to define the types of arguments the C function requires are:
| **name** | The name by which Caché calls the C function. This name is the first argument of the $ZF function. |
| **argtypes** | A string that describes how the arguments are passed. It can contain zero or more of the following characters, placed between quotation marks: |
| | • i—an integer |
| | • p—a pointer to an integer |
| | • c—a pointer to a zero-terminated character string (8-bit characters) |
| | • f—a pointer to a floating-point number |
| | • d—a pointer to a double-precision number |
| | • v—an OpenVMS fixed length string descriptor |
| | • b—a counted byte array (see the reference page for $ZF for information on zarray) |
| | • t—a translated string (in counted byte array) |
| | • s—a counted wide string (16-bit characters, see the reference page for $ZF for information on zwarray) |
| | • w—a zero-terminated wide character string (16–bit characters) |

For example, “ip” indicates an integer and a pointer to an integer. If you use an uppercase “P,” “C,” “F,” “D,” or “V,” you indicate that the argument is also an output argument. This means the routine will return a value in that Caché variable. If you specify more than one output argument, $ZF returns the arguments as a string delimited by commas.

Structure and argument prototype definitions can be seen in the include file cdzf.h. Additional InterSystems internal definitions are also declared there.

**Note:** Due to normal C calling conventions, arguments that are passed as integers cannot be output arguments.

| **entry-point** | The entry point name of the user-written C function. |

**UZF.C Example**

This C function translates strings into uppercase. It is stored in the file upper.c:

---

Using Caché ObjectScript

140
mupper(string)
    char *string;
{
    while (*string)
    {
        if (*string >= 'a' && *string <= 'z')
            *string += 'A' - 'a';
        ++string;
    }
    return(0);
}

The following ZFENTRY statement describes this C function:

ZFENTRY("UPPER", "C", mupper)

Caché calls this upper.c as follows:

Set LOW = "abcdef"
Set HIGH = $ZF("UPPER", LOW)

After the preceding code executes, LOW still contains “abcdef” and HIGH contains “ABCDEF”.

A second C function, stored in the file sum3.c, returns the sum of three numbers:

sum3(a,b,c,d)
    float *a, *b, *c
    double *d;
{
    *d = *a + *b + *c;
    return(0)
}

The following ZFENTRY statement describes this C function:

ZFENTRY("ADDTHREE", "fffD", sum3)

Caché calls this function as follows:

Set SUM = $ZF("ADDTHREE", 50, 42, 76, V)

The uzf.c file for the two functions in the two examples would be:

#include "cdzf.h"
#include "upper.c"
#include "sum3.c"
extern int mupper();
extern int sum3();

ZFBEGIN
ZFENTRY("UPPER", "C", mupper)
ZFENTRY("ADDTHREE", "fffD", sum3)

Examine the file CZF.C supplied with Caché for other examples.
12.2.2.5 Step 4: Compile the $ZF Function Descriptions

To compile uzf.c on UNIX, use the command:

```
c -c uzf.c
```

To compile uzf.c on Windows, use the command:

```
c1 -c uzf.c
```

This produces an output file called uzf.obj in the private directory that you can link to Caché.

12.2.2.6 Step 5: Link $ZF Functions into Caché

To link the C functions for use with $ZF (on UNIX):

1. Either perform your build in the `<install-dir>/source/Cache` directory, or make a private directory of your own with a copy of the required files. You are linking together the following files:
   - cache.o (provided with the Caché distribution kit)
   - main.o (provided with the Caché distribution kit)
   - shdir.o (provided with the Caché distribution kit)
   - uzf.o (created by you in your private namespace)

2. Modify the clink file in the build directory. Replace the file czf.o with uzf.o.

3. Run clink. If you get the message “Unresolved externals,” you may need to include other libraries; call InterSystems if you need help in determining which files to include. Note that if you place the C functions in separate files, you need to compile them separately and declare them in uzf.c, but not include them there. You then edit clink to contain uzf.o as well as other .o files.

4. Save the version of Caché that is in `/cachesys/bin/` (or wherever your installation directory may be) so that you can restore it, if necessary.

To link the C functions for use with $ZF (on Windows):

1.Either perform your build in the `<install-dir>/source/Cache` directory (where `<install-dir>` is the root directory for your configuration), or make a private directory of your own with a copy of the required files. You are linking together the following files:
   - cache.obj
   - main.obj
• shdir.obj
• clink.bat.

You link them with the file uzf.obj that you created by compiling uzf.c in your private directory.

2. Modify the clink.bat file in the build directory.

3. Replace the file czf.obj with uzf.obj.

4. Run clink. This produces the file cache.exe in your private directory. If you keep the C functions in separate files, you need to compile them separately and declare them in uzf.c, but not include them there. If you get the message “Unresolved externals...,” you may need to include other libraries. You then edit clink.bat to contain uzf.obj as well as other .obj files. Call InterSystems Worldwide Response Center if you need help in determining which files to include.

5. Save the current installed version of cache.exe that is in <install-dir>/bin (where <install-dir> is the root directory for your configuration), so you can restore it.

12.2.2.7 Step 6: Replace the Caché Executable

To replace the Caché executable for UNIX:

1. Shut down Caché by calling

   ccontrol stop <configname>

   where <configname> is the name of the Caché installation.

2. Copy the file cache from the private directory to <install-dir>/bin/, where <install-dir> is the root directory for your configuration.

3. Link cutil and cache by entering the following commands:

   # ./clink
   # cp cache <install-dir>/bin/

   where <install-dir> is the root directory for your configuration.

To reinstall Caché for Windows:

1. Stop Caché using the Stop Caché choice from the cube menu.

2. Copy the cache.exe file from your private directory to <install-dir>/bin, where <install-dir> is the root directory for your configuration.
When you restart Caché, the functions are available.

12.2.2.8 Step 7: Restart Caché

On UNIX, start up Caché with the cstart script or by calling:

ccontrol start <configname>

where <configname> is the installation of Caché to start.

On Windows, start Caché from the Caché Cube.

You now have a version of Caché that includes your C functions.

Returning to the Original Version of Caché

To return to the original version of Caché in the Caché system manager's directory (on UNIX):

1. Shut down Caché.
2. Copy the original saved cache back into <install-dir>/bin, where <install-dir> is the root directory for your configuration.

To return to the original version of Caché in the Caché manager's directory (on Windows):

1. Shut down Caché.
2. Copy the original saved cache.exe into <install-dir>/bin, where <install-dir> is the root directory for your configuration.

12.2.3 $ZF Calls Support Byte Arrays

Using the $ZF function, you can pass and receive byte arrays (also called “counted strings”) to and from user-written C routines. Declare byte arrays as having the type ZARRAYP. Variables of this type are pointers to the following C structure:

```c
typedef struct zarray {
    unsigned short len;
    unsigned char data[1]; /* 1 is a dummy value */
} *ZARRAYP;
```

This structure contains two fields:

- An unsigned short that contains the length of the byte array
- An unsigned character array that contains the byte data
For example, this routine copies a number of bytes from a buffer into a ZARRAYP structure:

```c
bytecpy(len, buf, bytestr)
unsigned short len;
unsigned char *buf;
ZARRAP bytestr;
{
    unsigned char *p, *q;
    p = buf;
    q = bytestr->data;
    bytestr->len = len;
    while(len--) *q++=*p++;
}
```

The maximum size of the array returned is the maximum string size, which is 32,767 bytes. The maximum total length of the arguments is the size of the string stack, about 64KB, depending on the type of the argument.

**CAUTION:** The string returned from $ZF$, which holds the return arguments, contains commas (,) as separators. Because commas can also be a part of byte arrays, declare these arrays at the end of the argument list and use one array per call.

### 12.3 Building Callout Modules as Dynamic Shared Libraries

You can build your callout routines as a DLL (on Windows), a shared library (on UNIX), or a shared executable (on OpenVMS), instead of linking them statically. On OpenVMS, this is optional, since CZF.EXE is a separate image anyway, but it is advantageous on UNIX and Windows, since it does not require rebuilding the Caché executable. The function-calling convention for a library is different than that of a linked callout module, however. For example:

```
$ZF("MYFUNC",args...)
```

might become:

```
$ZF(-3,"mylibrary","MYFUNC",args...)
```

There are several call conventions, each of which suits different requirements. The shared library or DLL build procedures are highly platform-specific. Check the compiler and linker documentation for the relevant options.
12.4 Referencing a User-Defined Callout Module

These descriptions apply to custom dynamic link libraries (Windows), dynamic shared libraries (UNIX), and shared executable images (OpenVMS). For simplicity, the term DLL is employed for all.

The callout module source is the same as that for a statically linked callout module, except that the define ZF_DLL is added before the include of header file cdzf.h (and also ccallin.h, if it applies). For example:

```
#define ZF_DLL
#include "ccallin.h"
#include "cdzf.h"
#include <stdio.h>
... user function declarations ...
```

The ZF_DLL define causes the ZFBEGIN macro to generate an internal function named GetZFTable, that is called when loading the DLL to return a pointer to its internal ZFENTRY (zfe) table, which is then used instead of the static zfe table to satisfy subsequent lookups of the names of user-written routines. Note that the symbol name “GetZFTable” must have been exported when linking the DLL.

12.4.1 Statically Linked Callouts

The following code specifies how to invoke statically linked callouts:

- Set `<result>=$ZF(<func_name>[,<args ...>])`
  
  Call the named function identified by `<func_name>` in the static czf module, passing the arguments if any. The result of the call is returned as the value of the function.

- Set `<result>=$ZF(<func_id>[,<args ...>])`
  
  If the sequence number `<func_id>` of the desired function within the ZFENTRY table is known (the first entry is number 1), it may be used instead of the function name for faster access.

12.4.2 A Single Active DLL: The $ZF(-3) Call Interface

The $ZF(-3) function can be used to load a single active Dynamic Link Library (DLL). On the initial reference, Caché attempts to load the specified DLL and obtain the address of the function GetZFTable within it. If a different DLL name is used later on, the current DLL is implicitly unloaded prior to loading the new one.
The relevant calls are:

- Set <result>=$ZF(-3,<dll_name>,<func_name>[,<args ...>])
  Load the named DLL if it isn't already loaded and call the named function identified by <func_name>, passing the arguments, if any. The result of the call is returned as the value of the $ZF(-3) function.

- Set <result>=$ZF(-3,"",<func_name>[,<args ...>])
  After the initial reference, the DLL name argument may be omitted for faster access.

As with static Callout modules, if the sequence number <func_id> of the desired function within the ZFENTRY table is known, it may be used instead of the function name for faster access.

To explicitly unload the current DLL, call:

Set <result>=$ZF(-3,"")

To simply load a DLL, call:

Set <result>=$ZF(-3,<dll_name>)

12.4.3 Multiple Simultaneous Dynamic Link Libraries

Support for this interface involves two functions:

- $ZF(-4) — Loads, unloads, and manages loaded DLLs.
- $ZF(-5) — Invokes a function in a loaded DLL.

These functions and $ZF(-3) coexist and can be used interchangeably in applications:

- Set <dll_id>=$ZF(-4,1,"<dll_name>")
  Load the named DLL if it isn't already loaded and return its <dll_id> for use with future $ZF(-4) or $ZF(-5) calls. Can generate a <STORE> error if space cannot be allocated. The <dll_id> is related to the internal address of the structure that describes that DLL and should not be maintained in persistent storage. If a DLL is unloaded and reloaded, the value can change.

- Set <result>=$ZF(-4,2)
  Unload all loaded DLLs (except for the DLL that might be loaded by $ZF(-3)) and return all allocated space to the system.

- Set <result>=$ZF(-4,2,<dll_id>)
Unload the specified DLL. The storage space is returned to the free pool for loading DLLs, it is not returned to the system.

- Set `<func_id>=$ZF(-4,3,<dll_id>,<func_name>)`

  Look up the named function in the specified loaded DLL and return its `<func_id>`.
  The `<func_id>` and the `<dll_id>` are the two pieces of information you need for `$ZF(-5)` to actually invoke a function in a loaded DLL. The `<func_id>` is the function index in the ZFENTRY table. This only changes if the DLL is modified and rebuilt. You do not need to use this function if you already knows the index of the function to call (such as if it is provided in `#define` statements in an ObjectScript macro include file).

- Set `<result>=$ZF(-5,<dll_id>,<func_id>[,<args>......])`

  Call the function identified by `<func_id>` in the DLL identified by `<dll_id>` passing the arguments if any. The result of the call is returned as the value of the `$ZF(-5)` function.

### 12.4.4 Referencing a DLL by Externally-Defined Index

Support for this interface involves additional functionality:

- `$ZF(-4)` — Manages system and process DLL index tables.
- `$ZF(-6)` — Invokes a function in a DLL by index.

where specific calls are:

- Set `<result>=$ZF(-6,<dll_index>[,<func_id>[,<args>......]])`

  Call the function identified by `<func_id>` in the DLL identified by `<dll_index>` passing the arguments if any. The first reference to `<dll_index>` causes `<dll_index>` to be searched for in the process DLL index table in order to load the associated DLL name. If `<dll_index>` is not found in the process DLL index table, the system DLL index table is searched. If it is not found in either table, an error is generated. The result of the call is returned as the value of the `$ZF(-6)` function.

At the present time, `<dll_index>` must be an integer > 0. It should be a unique identifier assigned by the user to serve as an index for the DLL file name.

The purpose of this call is to provide a fast DLL function interface that does not require the program to cache a `<dll_id>`, by substituting it with a user defined `<dll_index>`. It also assumes that the program manages each `<func_id>` so they don't need to be looked up by `<func_name>` and cached using `$ZF(-4,3)`. Thus no initialization pass is required for the program to load the DLL and obtain its ID and those of its function entries.
This is effectively a high-performance substitute for $ZF(-5,$ZF(-4,1,"<dll_name>"),<func_id>[,<args>.....]).

If the function ID and arguments are omitted, this call just does a load test of the DLL—it verifies the validity of the DLL index and loads the image. The image location is returned.

- Set <result>=$ZF(-4,4,<dll_index>)

Unload the specified DLL by index. This is a substitute for $ZF(-4,2,$ZF(-4,1,"<dll_name>"))

- Set <result>=$ZF(-4,5,<dll_index>,<dll_name>)

Create an entry in the system DLL index table, to uniquely associate <dll_index> with <dll_name>. If there is already an entry for <dll_index>, it is replaced by the new definition. You can create aliases for the same <dll_name>, but a process can only use one at a time. This is a privileged (manager) call. InterSystems currently reserves index numbers 1024 to 2047 for its own use.

- Set <result>=$ZF(-4,6,<dll_index>)

Delete the entry for <dll_index> in the system DLL index table, if one exists. This is a privileged (manager) call.

- Set <result>=$ZF(-4,7,<dll_index>,<dll_name>)

Create an entry in the process DLL index table, to uniquely associate <dll_index> with <dll_name>. This is analogous to the $ZF(-4,5,<dll_index>,<dll_name>) call except that the entry is made in a process private data structure. This table is searched before the system table, so it can be used to override system-wide definitions.

- Set <result>=$ZF(-4,8,<dll_index>)

Delete the entry for <dll_index> in the process DLL index table, if one exists.

- Set <result>=$ZF(-4,8)

Delete all entries in the process DLL index table.

### 12.4.5 DLL Runup and Rundown Functions

When a DLL (shared object, shared image) is first loaded, an internal runup function named ZFInit—if one has been defined—is called. No arguments are passed. The return code from this function should be zero to indicate absence of error, or non-zero to indicate some problem. If the call was successful, the address of the rundown function (see below)—if one has been defined—is saved. The runup call will occur on the first reference to:
When a DLL is unloaded, an internal rundown function named \texttt{ZFUnload}—if one has been defined, and if \texttt{ZFInit} either was not defined or its invocation was successful—is called. No arguments are passed. It will not however be called at process halt. The rundown call will occur on the first reference to:

- \texttt{ZF(-3,<dll_name>)} to load a new DLL
- \texttt{ZF(-4,2,<dll_id>)} to unload this DLL
- \texttt{ZF(-4,2)} to unload all DLL's
- \texttt{ZF(-4,4,<dll_index>)}

If some error occurs during the rundown function, further calls to it will be disabled, to allow unloading of the DLL. The return value from \texttt{ZFUnload} is currently ignored.

When building the DLL, you may need to explicitly export the symbols \texttt{ZFInit} and \texttt{ZFUnload} during the link procedure.

### 12.5 Sample Caché Callout DLL Build Procedures

Compile and link options may differ among versions of operating systems and development tools, so you should consult vendor documentation. The following example is for a single C source file named \texttt{mycallout.c}.

#### 12.5.1 Windows

The definition file is \texttt{mycallout.def}:

\begin{verbatim}
LIBRARY mycallout
DESCRIPTION 'ZF Callout Module'
EXPORTS
  GetZFTable @1
  ZFInit @2    // optional
  ZFUnload @3   // optional
\end{verbatim}
The code to build it is:

> cl /c mycallout.c
> link /dll mycallout.obj /def:mycallout.def

12.5.2 Linux

The code to build it is:

$ gcc -c mycallout.c
$ gcc mycallout.o -shared -o mycallout.so

12.5.3 Tru64 UNIX

The code to build it is:

$ cc -shared -c mycallout.c
$ cc mycallout.o -shared -o mycallout.so

12.5.4 HP 32-bit

The code to build it is:

$ cc +DA1.0 +DS1.0 +z -c mycallout.c
$ ld mycallout.o -b -o mycallout.so

12.5.5 HP 64-bit

The code to build it is:

$ cc +DA2.0W +DS2.0W +z -c mycallout.c
$ ld mycallout.o +DA2.0W -b -o mycallout.so

12.5.6 PowerPC

The definition file is mycallout.exp:

GetZFTable
ZFInit            # optional
ZFUnload          # optional

The code to build it is:

$ cc -c mycallout.c
$ ld -o mycallout.so mycallout.o -bE:mycallout.exp -bM:SRE -bnoentry -lc
12.5.7 OpenVMS

The code to build it is:

```
> cc mycallout.c  
> link/shareable=mycallout.exe sys$input/options
cluster=mycallout,,,mycallout
gsmatch=always,0,0
symbol_vector=(GetZFTable=proc)
symbol_vector=(ZFInit=proc)    ! optional
symbol_vector=(ZFUload=proc)  ! optional
```

12.6 Reading and Writing to Devices Opened in UNIX

You can use $ZF to issue I/O commands to devices you open outside of Caché on UNIX systems. Take the following steps:

1. Open a file descriptor at the operating system level.
2. Open the same file descriptor as a device within Caché.
3. Read and write to the device.
4. Close it when you are finished.

You cannot use file descriptor devices with the DEVICE utility. You also cannot use them to open devices opened in Caché under a different name, such as device 47 and open Caché database files.

12.6.1 Opening the File Descriptor

To open the file descriptor:

1. Open a file descriptor in UNIX, using the $ZF function.
2. Issue an Open command in Caché to the file descriptor. Use the following form, in which file_descriptor is an integer that identifies the file descriptor that you opened previously in UNIX:

```
Open |FD| file_descriptor
```
After the device is open, you can issue read and/or write commands to it, depending on how you opened the file at the UNIX level.

Caché treats file descriptor devices as sequential files. Thus, they can have only the parameters appropriate to sequential files. Caché ignores the read-only and write-only parameters. In other words, if you try to read a file that is opened for write-only at the UNIX level, Caché produces a <READ> error even if you opened the file in Caché with the “RW” parameters. Caché also ignores the “A” (append) and “N” (new) parameters.

### 12.6.2 Closing the File Descriptor

You must issue an explicit Close command in Caché for the file descriptor device.

The Close option, “K”, closes the file descriptor device at the Caché level only. K does not close the file descriptor device at the UNIX level. If you omit the K option, Close closes the device first at the Caché level and then at the UNIX level.

Caché ignores all other Close parameters when you close a file descriptor device.

**CAUTION:** You must close the file descriptor device at the Caché level before you close it at the UNIX level. Closing file descriptor devices by issuing Close without the K parameter does this automatically. If you close a file descriptor device in UNIX without first closing it in Caché, you risk database and other device corruption.

This example shows how you can use file descriptors that the process has opened at the UNIX level:

```objectscript
USER> Open "cache.test":"WNS" ; create UNIX file
USER> Use "cache.test" W "this is a test",! ; enter text into file
USER> Close "cache.text" ; close the file
USER> Write $ZF("OPENFD","cache.test") ; write file with $ZF function
4
USER> Open "|FD|4" ; 4 is the file descriptor from the previous command
USER> Use "|FD|4" Read X
USER> Write X
this
USER> Close "|FD|4":"k"
```

To keep the device open in UNIX, use the following for the last line in the previous example:

```objectscript
CLOSE "|FD|4":"K"
```

The Close command with the K parameter lets you reopen the device in Caché without opening it again in UNIX and continue reading from where you left off.
Caché offers a callin interface you can use from within C programs to execute Caché commands and evaluate Caché expressions. This chapter describes this interface and includes the following sections:

- Callin Function Overview
- About ASCII and Unicode Functionality
- Other Information and Requirements
- Compiling, Linking, and Running a callin Program

For reference material on the callin interface, see the Callin Function Reference.

You can also use this callin interface with $ZF routines you write. You must use Caché ObjectScript to take advantage of the callin feature.

The callin interface also supports Unicode characters and translations between encodings.

You can use the callin interface in one of two ways:

- Create a callin executable by linking code together with the Caché engine functions. The program calls CacheStart to establish a connection with Caché and its facilities, at which time the process becomes integrated with the Caché runtime environment and other Caché processes. After establishing the connection, you can use any of the other operations available according to your needs. When the connection is no longer needed, the program calls CacheEnd, at which time the process exits the Caché runtime environment and the functions of the Caché engine are no longer accessible.

- The user code is invoked from a running Caché process via callout. The code may either be statically linked or a dynamic load library. The callin functions are available, but no
calls to **CacheStart** or **CacheEnd** are required. This form of utilization is known as *callback*.

The callin interface permits a wide variety of applications. For example, you can use it to make Caché ObjectScript available from an integrated menu or GUI. If you gather information from an external device, such as an Automatic Teller Machine or piece of laboratory equipment, the callin interface lets you store this data in a Caché database.

Although Caché currently supports only C and C++ programs, any language that uses the calling standard for that platform (OpenVMS, UNIX, Windows) can invoke the callin functions.

The distribution media contains a sample C program, `ccallin.c`, which demonstrates the major callin interface functions. To learn more about how to compile, link, and execute this program, see the later section, Using `ccallin.c`.

### 13.1 Callin Function Overview

The Caché callin interface consists of ASCII (8-bit) and Unicode (16-bit) functions. Most of these functions call into Caché. Some of these functions act without entering Caché.

`ccallin.h` defines prototypes for these functions, which allows your C compiler to test for valid parameter data types when you call these functions within your program. (`ccallin.h` also defines data types for function parameters.)

There are two groups of functions: high-level application functions, and low-level primitives. There is reference material for all the callin functions in the Callin Reference and the callin functions are summarized in the following tables. The tables note whether or not each function calls into Caché.

#### Caché Callin Interface Functions, Part I (High-Level Functions)

<table>
<thead>
<tr>
<th>High-Level Function</th>
<th>Calls into Caché</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CacheAbort</strong></td>
<td>No</td>
<td>Tells Caché to terminate the current request when convenient via <strong>RESJOB</strong> or <strong>CTRL-C</strong>.</td>
</tr>
<tr>
<td><strong>CacheContext</strong></td>
<td>No</td>
<td>Returns an integer indicating whether you are in a <strong>$ZF</strong> callback session, in the Caché side of a callin call, or in the user program side.</td>
</tr>
<tr>
<td>High-Level Function</td>
<td>Calls into Caché</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CacheConvert</td>
<td>No</td>
<td>Returns the value of the Caché expression returned by CacheEval.</td>
</tr>
<tr>
<td>CacheCtrl</td>
<td>No</td>
<td>Performs miscellaneous control functions.</td>
</tr>
<tr>
<td>CacheCvtInA</td>
<td>No</td>
<td>Translates string with specified external character set encoding to the ASCII character string encoding used internally in Caché 8-bit version.</td>
</tr>
<tr>
<td>CacheCvtInW</td>
<td>No</td>
<td>Translates string with specified external character set encoding to the Unicode character string encoding used internally in the Caché Unicode version.</td>
</tr>
<tr>
<td>CacheCvtOutA</td>
<td>No</td>
<td>Translates string from the ASCII character string encoding used internally in the Caché 8-bit version to a string with the specified external character set encoding.</td>
</tr>
<tr>
<td>CacheCvtOutW</td>
<td>No</td>
<td>Translates string from the Unicode character string encoding used internally in the Caché Unicode version to a string with the specified external Unicode set encoding.</td>
</tr>
<tr>
<td>CacheEnd</td>
<td>Yes</td>
<td>Terminates a Caché session and, if necessary, cleans up a broken connection.</td>
</tr>
<tr>
<td>CacheErrorA</td>
<td>No</td>
<td>Returns the most recent error message, its associated source string, and the offset to where in the source string the error occurred.</td>
</tr>
<tr>
<td>CacheErrorW</td>
<td>No</td>
<td>Unicode string counterpart to CacheErrorA.</td>
</tr>
<tr>
<td>CacheErrxlateA</td>
<td>No</td>
<td>Returns the Caché error string associated with error number returned from a callin function.</td>
</tr>
<tr>
<td>CacheErrxlateW</td>
<td>No</td>
<td>Unicode string counterpart to CacheErrxlateA.</td>
</tr>
<tr>
<td>CacheEvalA</td>
<td>Yes</td>
<td>Evaluates a Caché ObjectScript expression.</td>
</tr>
<tr>
<td>CacheEvalW</td>
<td>Yes</td>
<td>Unicode string counterpart to CacheEvalA.</td>
</tr>
<tr>
<td>CacheExecuteA</td>
<td>Yes</td>
<td>Execute a Caché ObjectScript command.</td>
</tr>
<tr>
<td>CacheExecuteW</td>
<td>Yes</td>
<td>Unicode string counterpart to CacheExecuteA.</td>
</tr>
</tbody>
</table>
For high-performance interfaces, there are the primitives listed below. Their use typically involves pushing call arguments one at a time onto an argument stack, invoking the desired operation, then popping the return value (if any) off the argument stack. The functions must be used correctly to avoid memory leaks or corruption of data structures in memory.

Where possible, pointers to data values in memory are passed instead of copying the data itself, so as a general rule each argument should be kept in a separate area that is preserved until the desired operation is executed. Similarly, a return value should either be used immediately or else be copied to a location that will not be modified until the value is no longer needed.

Function names without a suffix are either not character oriented, or are versions for 8-bit character strings. Function names with a “W” suffix are versions for wide (16-bit, or Unicode) character strings.

**Caché Call-In Interface Functions, Part II (Low-Level Primitives)**

<table>
<thead>
<tr>
<th>Low-Level Primitive</th>
<th>Calls into Caché</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheCloseOref</td>
<td>No</td>
<td>Decrement the reference counter for an OREF</td>
</tr>
<tr>
<td>CacheDoFun</td>
<td>Yes</td>
<td>Perform a routine call (special case)</td>
</tr>
<tr>
<td>CacheDoRtn</td>
<td>Yes</td>
<td>Perform a routine call</td>
</tr>
<tr>
<td>CacheExtFun</td>
<td>Yes</td>
<td>Perform an extrinsic function call</td>
</tr>
<tr>
<td>Low-Level Primitive</td>
<td>Calls into Caché</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>CacheGetProperty</td>
<td>Yes</td>
<td>Obtain the value for a property</td>
</tr>
<tr>
<td>CacheGlobalGet</td>
<td>No</td>
<td>Obtain the value for a global reference</td>
</tr>
<tr>
<td>CacheGlobalSet</td>
<td>No</td>
<td>Store the value for a global reference</td>
</tr>
<tr>
<td>CacheIncrementCountOref</td>
<td>No</td>
<td>Increment the reference counter for an OREF</td>
</tr>
<tr>
<td>CacheInvokeClassMethod</td>
<td>Yes</td>
<td>Perform a class method call</td>
</tr>
<tr>
<td>CacheInvokeMethod</td>
<td>Yes</td>
<td>Perform an instance method call</td>
</tr>
<tr>
<td>CachePop</td>
<td>No</td>
<td>Pop a value off argument stack</td>
</tr>
<tr>
<td>CachePopCvtW</td>
<td>No</td>
<td>Pop a local string off argument stack and translate it to Unicode</td>
</tr>
<tr>
<td>CachePopDb1</td>
<td>No</td>
<td>Pop value off argument stack and convert it to a double</td>
</tr>
<tr>
<td>CachePopInt</td>
<td>No</td>
<td>Pop a value off argument stack and convert it to an integer</td>
</tr>
<tr>
<td>CachePopList</td>
<td>No</td>
<td>Pop a $LIST object off argument stack and translate it</td>
</tr>
<tr>
<td>CachePopOref</td>
<td>No</td>
<td>Pop an OREF off argument stack</td>
</tr>
<tr>
<td>CachePopPtr</td>
<td>No</td>
<td>Pop a pointer value off argument stack</td>
</tr>
<tr>
<td>CachePopStr</td>
<td>No</td>
<td>Pop a value off argument stack and convert it to a string</td>
</tr>
<tr>
<td>CachePopStrW</td>
<td>No</td>
<td>Wide version of CachePopStr</td>
</tr>
<tr>
<td>CachePushClassMethod</td>
<td>No</td>
<td>Push a class method reference onto argument stack</td>
</tr>
<tr>
<td>CachePushClassMethodW</td>
<td>No</td>
<td>Wide version of CachePushClassMethod</td>
</tr>
<tr>
<td>CachePushCvtW</td>
<td>No</td>
<td>Translate a Unicode string to local and push it onto argument stack</td>
</tr>
<tr>
<td>CachePushDb1</td>
<td>No</td>
<td>Push a double onto argument stack</td>
</tr>
<tr>
<td>CachePushFunc</td>
<td>No</td>
<td>Push a function reference onto argument stack</td>
</tr>
<tr>
<td>Low-Level Primitive</td>
<td>Calls into Caché</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CachePushFuncW</td>
<td>No</td>
<td>Wide version of CachePushFunc</td>
</tr>
<tr>
<td>CachePushFuncX</td>
<td>No</td>
<td>Push an extended function reference onto argument stack</td>
</tr>
<tr>
<td>CachePushFuncXW</td>
<td>No</td>
<td>Wide version of CachePushFuncX</td>
</tr>
<tr>
<td>CachePushGlobal</td>
<td>No</td>
<td>Push a global name onto argument stack</td>
</tr>
<tr>
<td>CachePushGlobalW</td>
<td>No</td>
<td>Wide version of CachePushGlobal</td>
</tr>
<tr>
<td>CachePushGlobalX</td>
<td>No</td>
<td>Push an extended global name onto argument stack</td>
</tr>
<tr>
<td>CachePushGlobalXW</td>
<td>No</td>
<td>Wide version of CachePushGlobalX</td>
</tr>
<tr>
<td>CachePushInt</td>
<td>No</td>
<td>Push an integer onto argument stack</td>
</tr>
<tr>
<td>CachePushList</td>
<td>No</td>
<td>Translate and push a $LIST object onto argument stack</td>
</tr>
<tr>
<td>CachePushMethod</td>
<td>No</td>
<td>Push an instance method reference onto argument stack</td>
</tr>
<tr>
<td>CachePushMethodW</td>
<td>No</td>
<td>Wide version of CachePushMethod</td>
</tr>
<tr>
<td>CachePushOref</td>
<td>No</td>
<td>Push an OREF onto argument stack</td>
</tr>
<tr>
<td>CachePushProperty</td>
<td>No</td>
<td>Push a property name onto argument stack</td>
</tr>
<tr>
<td>CachePushPropertyW</td>
<td>No</td>
<td>Wide version of CachePushProperty</td>
</tr>
<tr>
<td>CachePushPtr</td>
<td>No</td>
<td>Push a pointer value onto argument stack</td>
</tr>
<tr>
<td>CachePushRtn</td>
<td>No</td>
<td>Push a routine reference onto argument stack</td>
</tr>
<tr>
<td>CachePushRtnW</td>
<td>No</td>
<td>Wide version of CachePushRtn</td>
</tr>
<tr>
<td>CachePushRtnX</td>
<td>No</td>
<td>Push an extended routine reference onto argument stack</td>
</tr>
<tr>
<td>CachePushRtnXW</td>
<td>No</td>
<td>Wide version of CachePushRtnX</td>
</tr>
<tr>
<td>CachePushStr</td>
<td>No</td>
<td>Push a string onto argument stack</td>
</tr>
<tr>
<td>CachePushStrW</td>
<td>No</td>
<td>Wide version of CachePushStr</td>
</tr>
</tbody>
</table>
### 13.1.1 How the Functions Operate

#### 13.1.1.1 Function Type

All callin functions return values of type `int`, where the return value does not exceed the capacity of a 16-bit integer. Returned values can be `CACHE_SUCCESS`, a Caché error, or a callin interface error.

#### 13.1.1.2 Error Handling

There are two types of errors:

- Caché errors — The return value of a Caché error is a positive integer.
- Interface errors — The return value of an interface error is 0 or a negative integer.

`ccallin.h` defines symbols for all Caché and interface errors including the interface errors `CACHE_SUCCESS` (0) and `CACHE_FAILURE` (-1). You can translate Caché errors (positive integers) by making a call to the callin function `CacheErrxlate`.

### 13.2 About ASCII and Unicode Functionality

Because a number of Caché callin functions operate on strings, these functions have both ASCII and Unicode versions. For some callin functions, there are function names, type definitions, and data type item values with a distinguishing “A” or “W” character in the name. These indicate whether they apply to ASCII or Unicode (Wide) characters.

For some functions, their allowed inputs and outputs vary, depending on whether they are running on an ASCII (8-bit) system or a Unicode (16-bit) system. For many of the “A” (ASCII) functions, the arguments are defined as accepting a `CACHESTR`, `CACHE_STR`, `CACHESTRP`, or `CACHE_STRP` type. These symbol definitions (without the “A” or “W” ) can conditionally be associated with either the ASCII or Unicode names, depending on

<table>
<thead>
<tr>
<th>Low-Level Primitive</th>
<th>Calls into Caché</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheSetProperty</td>
<td>Yes</td>
<td>Store the value for a property</td>
</tr>
<tr>
<td>CacheUnPop</td>
<td>No</td>
<td>Restore the stack entry from CachePop</td>
</tr>
</tbody>
</table>
whether the symbol CACHE_UNICODE is defined at compile time. This way, you can write source code with neutral symbols that works with either ASCII or Unicode encodings. InterSystems recommends that, for best performance, you use the kind of string native to your installed version of Caché.

The following excerpt from ccallin.h illustrates the concept:

```
#ifdef CACHE_UNICODE /* Unicode char strings */
#define CACHE_STR CACHEWSTR
#define CACHE_STRP CACHEWSTRP
#define CACHE_STRING CACHE_WSTRING
...
#define CacheStart CacheStartW
...
#else /* ASCII character strings */
#define CACHE_STR CACHE_ASTR
#define CACHE_STRP CACHE_ASTRP
#define CACHE_STRING CACHE_ASTRING
...
#define CacheStart CacheStartA
...
#endif
```

The following table describes the complete set of names:

<table>
<thead>
<tr>
<th>Alternate Names for Functions and Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral Name</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>CACHESTR</td>
</tr>
<tr>
<td>CACHESTRP</td>
</tr>
<tr>
<td>CACHE_STR</td>
</tr>
<tr>
<td>CACHE_STRP</td>
</tr>
<tr>
<td>CACHE_STRING</td>
</tr>
<tr>
<td>CacheCvtIn</td>
</tr>
<tr>
<td>CacheCvtOut</td>
</tr>
<tr>
<td>CacheError</td>
</tr>
<tr>
<td>CacheErrxlate</td>
</tr>
<tr>
<td>CacheEval</td>
</tr>
<tr>
<td>CacheExecute</td>
</tr>
<tr>
<td>CachePrompt</td>
</tr>
<tr>
<td>CacheStart</td>
</tr>
</tbody>
</table>
13.2.1 Data Types

The callin interface supports both 8-bit and 16-bit functions, where this is relevant.

13.2.1.1 8-Bit Data Types

Caché supports the following Unicode-related data types:

• CACHE_ASTR — 8-bit counted string
• CACHE_ASTRP — Pointer to an 8-bit counted string

The type definition for these is:

```
#define CACHE_MAXSTRLEN 32767
typedef struct {
    unsigned short len; char str[CACHE_MAXSTRLEN];
} CACHE_ASTR, *CACHE_ASTRP;
```

The CACHE_ASTR and CACHE_ASTRP structures contain two elements:

• `len` — An integer. When used as input, this element specifies the actual length of the string whose value is supplied in the `str` element. When used as output, this element specifies the maximum allowable length for the `str` element; upon return, this is replaced by the actual length of `str`.

• `str` — A input or output string.

`CACHE_MAXSTRLEN` is the maximum length of a string that is accepted or returned. A parameter string need not be of length `CACHE_MAXSTRLEN` nor does that much space have to be allocated in the program.

13.2.1.2 Unicode Data Types

Caché supports the following Unicode-related data types:

• CACHEWSTR — Unicode counted string
• CACHEWSTRP — Pointer to Unicode counted string

The type definition for these is:

```
typedef struct {
    unsigned short len; unsigned short str[CACHE_MAXSTRLEN];
} CACHEWSTR, *CACHEWSTRP;
```

The CACHEWSTR and CACHEWSTRP structures contain two elements:
Callin Interface

- **len** — An integer. When used as input, this element specifies the actual length of the string whose value is supplied in the **str** element. When used as output, this element specifies the maximum allowable length for the **str** element; upon return, this is replaced by the actual length of **str**.

- **str** — A input or output string.

**CACHE_MAXSTRLEN** is the maximum length of a string that is accepted or returned. A parameter string need not be of length **CACHE_MAXSTRLEN** nor does that much space have to be allocated in the program.

On Unicode-enabled versions of Caché, there is also the data type **CACHE_WSTRING**, which represents the native string type. **CacheType** returns this type. Also, **CacheConvert** can specify **CACHE_WSTRING** as the data type for the return value; if this type is requested, the result is passed back as a counted Unicode string in a **CACHEWSTR** buffer.

### 13.3 Other Information and Requirements

#### 13.3.1 ccallin.h Contents

ccallin.h contains definitions of parameter values you use in your calls. You can add this file to the list of #include statements in your C program:

```c
#include "ccallin.h"
```

You can translate the distributed header file, ccallin.h. However, ccallin.h is subject to change and you must track any changes if you create a translated version of this file. InterSystems Worldwide Support Center does not handle calls about unsupported languages.

#### 13.3.2 Using Callin Functions in $ZF Routines

For instructions on compiling, linking, and running a **$ZF** function, see Calling Out of Caché.

Since you are already in Caché when you are running a **$ZF** routine, you do not need to call the functions **CacheStartA** and **CacheEnd**.
13.3.3 Requirements for Callin Programs

Your external program must follow certain rules to avoid corrupting Caché data structures. Corruption can cause a system hang.

13.3.3.1 Opening Files

Your program must ensure that it does not open so many files that it prevents Caché from opening the number of databases or other files it expects to be able to. Normally, Caché looks up the user's open file quota and reserves a certain number of files for opening databases, allocating the rest for the `Open` command. Depending on the quota, Caché expects to have between 6 and 30 Caché database files open simultaneously, and from 0 to 36 files open with the `Open` command.

13.3.3.2 Do Not Disable Interrupt Delivery on UNIX

UNIX uses interrupts. Do not prevent delivery of interrupts.

13.3.3.3 Call CacheEnd After CacheStartA Before Halting

If your Caché connection was established by a call to `CacheStartA`, then you must call `CacheEnd` when you are done with the connection. You can make as many callin function calls in between as you wish.

You must call `CacheEnd` even if the connection was broken. The connection can be broken by a call to `CacheAbort` with the `RESJOB` parameter.

`CacheEnd` performs cleanup operations which are necessary to prepare for another call to `CacheStartA`. Calling `CacheStartA` again without calling `CacheEnd` (assuming a broken connection) will return the code `CACHE_CONBROKEN`.

13.3.3.4 Exiting Your Program

If you are going to exit your program, you must be certain Caché ObjectScript has completed any outstanding request. Use the callin function `CacheContext` to determine whether you are within Caché ObjectScript. This call is particularly important in exit handlers and `Ctrl-C` or `Ctrl-Y` handlers. If `CacheContext` returns a non-zero value, you can invoke `CacheAbort`. 
13.3.3.5 OpenVMS Requirements

Terminating Callin Programs on OpenVMS
If you terminate a callin program by issuing an OpenVMS STOP/ID command, a harmless ghost process may result.

If OpenVMS receives the STOP/ID while the program is executing as a process in Caché, the command is ignored, and you receive a message indicating that the process cannot be deleted.

If the program is not a Caché process when the STOP/ID arrives, however, it has no such protection. Such processes terminate immediately and leave what looks like a ghost process in %SS. There is no risk of a system hang, however, as these processes do not own any Caché resources when they terminate.

Assigning Event Flags on OpenVMS
OpenVMS reserves certain local event flag numbers. To avoid using the same event flag numbers, your programs must use LIB$GETEF, a function that returns event flag numbers, to assign these numbers.

Exit Handlers Must Return to OpenVMS
In the OpenVMS environment, exit handlers must return and let the other declared exit handlers run to completion.

Use of ASTs
OpenVMS uses asynchronous traps (AST). Do not prevent delivery of ASTs.

Functions that call into Caché cannot be called from an AST; those that make calls exclusively within the client can use ASTs.

13.4 Compiling, Linking, and Running a Callin Program

The process for using the callin interface is:

1. Compile your program.
2. Link the ccاليin.c object, as described below.
3. Make sure the appropriate Caché configuration is running.
4. Run your executable.

Other, related topics are discussed in their various sections below.

### 13.4.1 Files Supplied

Files on your distribution medium that pertain to the Caché callin interface include the following:

- `ccallin.h` — header file with callin data structures and function prototypes
- `cdzf.h` — header file for callout, used in conjunction with `ccallin.h` for building callback modules (that is, dynamic load modules that were initially invokes from Caché via callout and that call back into Caché via the callin facilities)
- `ccallin.c` — callin sample routine

For OpenVMS

- `CLINK.OPT` — linker options specifications

For UNIX:

- `clink` — link script (also used for linking callout)
- `cache.o` — Caché engine functions
- `shdir.c` — specification of manager's directory
- `shdir.o` — location for the configuration being built
- `czf.o` — default callout sample module

For Windows:

- `clink.bat` — link script (also used for linking callout)
- `cache.obj` — Caché engine functions
- `shdir.c` — specification of manager's directory
- `shdir.obj` — location for the configuration being built
- `czf.obj` — default callout sample module

**Note:** If the customer callin code references special libraries, the appropriate clink file may need editing to include them.
13.4.2 Compiling, Linking, and Running on Windows

To compile, link, and run your callin program on Windows:

1. Make sure Caché is running. This step is necessary for you to be able to run your callin program.
2. Compile your program by entering the command:
   
   ```
   C:\> cl -c mycallin
   ```

3. Edit `shdir.c` to contain your manager’s directory, which, by default, is `c:\\cachesys\\mgr`. Then compile it:

   ```
   C:\> cl -c shdir.c
   ```

4. Link the `ccallin.c` object with the supplied callin file, `clink.bat`, by entering:

   ```
   C:\> clink mycallin.obj executable
   ```

   `clink.bat` uses the following syntax:

   ```
   clink [objectfile [executable]]
   ```

   where

   - `objectfile` — The name of the object file for your callin program. If you specify an object file of `main.o` or an object file of the empty string ("" or "), `clink` builds a callout executable.
   - `executable` — The name that you want for the final executable. If you omit a name, Caché assigns the default name “ccallin”

   If you do not supply a value for either, `clink` builds the callout executable.

5. Run the executable by entering:

   ```
   C:\> mycallin
   ```

13.4.3 Compiling, Linking, and Running on UNIX

To compile, link, and run your callin program on UNIX:

1. Make sure Caché is running. This step is necessary for you to be able to run your callin program.
2. Compile your program by entering the command:

   $ cc -c mycallin

3. Edit shdir.c to contain your manager's directory, which, by default, is /cachesys/mgr. For example:

   char shdir[64] = "/cachesys/mgr";

4. Then compile it:

   $ cc -c shdir.c

5. Link the ccallin.c object with the supplied callin file, clink, by entering:

   $ clink mycallin.o executable

   **clink** uses the following syntax:

   clink [objectfile [executable]]

   where

   • **objectfile** — The name of the object file of your callin program; by default, Caché names the executable “ccallin”. If you specify an object file of main.o or an object file of the empty string ("" or "), **clink** builds a callout executable and, by default, gives it a filename of cache.

   • **executable** — The name that you want for the final executable, where possible default names are determined by the **objectfile** argument.

   If you do not supply a value for either argument, **clink** builds the **callout** executable.

6. Run the executable by entering:

   $ mycallin

13.4.4 Compiling, Linking, and Running on OpenVMS

To compile, link, and run your callin program on OpenVMS:

1. Make sure Caché is running. This step is necessary as this procedure accesses some of the image files.

2. Compile your program by entering the command:

   $ CC MYCALLIN
3. Link the `callin.c` object with the supplied callin file, CLINK.OPT, by entering:

   $ LINK MYCALLIN,CLINK/OPT,SYS$LIBRARY:VAXCRTL/LIB

   or

   $ LINK/EXE=MYCALLIN SYS$INPUT/OPTION MYCALLIN.OBJ
   SYS$LIBRARY:VAXCRTL/LIB CACHESYS:CKERNAL/SHARE <CTRL-Z>

   For more information on using CLINK.OPT, see the section below.

4. Run the executable by entering:

   $ RUN MYCALLIN

### 13.4.4.1 Using CLINK.OPT

Use CLINK.OPT with the LINK command for OpenVMS systems. CLINK.OPT is a linker options file. It contains the lines:

```
CACHE.EXE/SHAREABLE
SYS$LIBRARY:VAXCRTL/LIBRARY
```

In order to support multiple simultaneous versions of Caché, a user-defined callin executable needs to know both where the appropriate CACHE.EXE is located and where the corresponding Caché manager's directory is located. Under the present structure, the executable is kept in the bin subdirectory beneath the installation directory and the manager's directory is the mgr subdirectory.

To successfully load and run a callin executable, define two process logicals:

- **CACHE** — contains the location of the CACHE.EXE file
- **CACHESYS** — contains the location of the manager's directory

For example:

```
DEFINE CACHE DKA0:[CACHESYS.BIN]CACHE.EXE
DEFINE CACHESYS DKA0:[CACHESYS.MGR]
```

For the default configuration or a machine with only one installation, you can use the system logicals that point to its files. System logical **CACHESYS** already gives the location of the manager's directory, so you need only provide the location of the executable:

```
DEFINE CACHE CACHEBIN:CACHE.EXE
```

If you invoke a callin program that has been linked against a different CACHE.EXE than the one specified by the **CACHE** logical, then unpredictable behavior may result.
To specify the location of the CACHE.EXE to link against, your options are to:

- Edit the distributed CLINK.OPT file to provide the location of the appropriate CACHE.EXE
- Specify the location on the command line for LINK

The following example assumes that CACHE.EXE is in DKA0:[CACHESYS.BIN] and that there is a single user source file called mycallin.c. To compile the source, the command is:

```bash
>CC MYCALLIN
```

To link the executable, the commands are:

```bash
>LINK/EXECUTABLE=MYCALLIN MYCALLIN,  SYS$INPUT:/OPTIONS DKA0:[CACHESYS.BIN]CACHE.EXE/SHAREABLE SYS$LIBRARY:VAXCRTL/LIBRARY <Ctrl-Z>
```

or else edit CLINK.OPT to read:

```plaintext
DKA0:[CACHESYS.BIN]CACHE.EXE/SHAREABLE SYS$LIBRARY:VAXCRTL/LIBRARY
```

and call LINK this way:

```bash
>LINK/EXECUTABLE=MYCALLIN MYCALLIN,  CLINK/OPTIONS
```

Due to 32/64–bit pointer conversion issues on OpenVMS, you should not build Caché callin or callout modules with 64–bit pointers, that is compiler option /pointersize=64. Use the default pointer size of 32 bits.

### 13.4.5 Compiling, Linking, and Running the ccallin.c Sample Program

ccallin.c is an interactive program that allows you to try some of the callin interface functionality. It follows the steps listed in this section. The copy of ccallin.c on the following pages includes numbers that correspond to these steps.

The first page contains the **main** routine; the following page contains the loop routine **cmdloop** and the display routine, **dispres**, called by **cmdloop** to print the result of a **CacheEval** call.

1. Asks you to set the parameters for **CacheStart**:
   ```plaintext
   Enter term flags (in hex):
   Enter timeout:
   Enter principal output device:
   Enter principal input device:
   ```

2. Starts a Caché process with **CacheStart**, using your parameters.
3. Enters a loop where it prompts you for input to either CacheExecute or CacheEvalA and then uses that input in a call to the appropriate function.

4. Displays a “:” prompt. At the prompt, you can enter a Caché command, such as:

   :Do ^%CD

   Alternatively, you can enter a Caché expression at the prompt. If you do this, you notify the program by first entering the equals “=” character; for example:

   :=^["[PRODDIR]""]PROD_REC

5. ccallin executes the command with a call to CacheExecute, printing any displayed information to the device you specified as your CacheStart principal output device parameter value. You will experience a normal %CD session in response to the line above.

   On OpenVMS, the response is:

   Directory: <RETURN>
   You're in directory AVM1$DKA100:[CACHESYS]

   On UNIX, the response is:

   Directory: /usr/cachesys

   On Windows, the response is:

   \\cachesys\test

6. You end the loop by typing the “~” character at the “:” prompt or by issuing the Caché Halt command.

7. When you end the loop, ccallin ends the Caché process by calling CacheEnd.

8. If you called ccallin from a $ZF function, you return to Caché. If you are not running from within a $ZF, ccallin prompts to see if you wish to restart from Step 1.

9. At the Restart? prompt, enter “Y” to return to Step 1 or any other character to end the program.

ccallin evaluates the expression with a call to CacheEval. If that call is successful, it then calls CacheType to get the type of value returned by CacheEval. It then calls CacheConvert to translate the value returned by CacheEval and print it to your specified output device.

For example, the value of the global sent above would be returned:

   result = /0/
13.4.5.1 Compiling and Linking ccallin.c

The procedure for this varies slightly by operating system.

**On Windows**

To compile, link, and run ccallin.c on Caché for Windows:

1. Compile the sample program:
   
   $ cl -c ccallin.c

2. Edit shdir.c to contain your manager's directory, which is `c:\cachesys\mgr` by default. Then compile it:
   
   $ cl -c shdir.c

3. Link the ccallin.o object with the Caché system code:
   
   $ clink ccallin.obj cext

4. Make sure Caché is running. Run the sample program:
   
   $ cext

**On UNIX**

To compile, link, and run ccallin.c on Caché for UNIX:

1. Make sure Caché is running.

2. Compile the sample program:
   
   $ cc -c ccallin.c

3. Link the ccallin.o object with the Caché system code:
   
   $ clink ccallin.o cext

4. Run the sample program:
   
   $ cext

**On OpenVMS**

To compile, link, and run ccallin.c on Caché for OpenVMS:

1. Make sure Caché is running.

2. Compile the sample program:
   
   $ CC ccallin.c
3. Link CCALIN.OBJ with the Caché system code:

```bash
$ LINK CCALIN,CLINK/OPT,SYS$LIBRARY:VAXCRTL/LIB
```

4. Run the sample program:

```bash
$ RUN CCALIN
```

### 13.4.5.2 Running ccallin

The following is an annotated `ccallin` session.

```bash
$ RUN CCALIN
Enter term flags (in hex): 1
Enter timeout: 30
Enter principal output device: <RETURN>
Enter principal input device: <RETURN>
: ^=TEMP ; Preceding the string with '=' results in a call to
; CacheEval. If the call is successful, result = /0/
; subsequent calls to CacheType and
; CacheConvert provide the value of ^TEMP,
; which is then printed. In this case, ^TEMP
; is currently set to 0. Since the following string is not
; preceded by a '=', CCALIN tries to process it as a
; Caché command by sending it to CacheExecute. Since it
; is not a valid command, Caché returns a <SYNTAX> error.
: ^TEMP<SYNTAX>
 ; The following two methods of making $ZU calls are valid
 ; inputs to CacheEval
: =$ZU(49,"[prod_dir]")
 result = /2,400,260,2,1,3,2,4,14,15,318,0,0/
 : Set X=$ZU(49,"[prod_dir]")
 : Write X
2,400,260,2,1,3,2,4,14,15,318,0,0
 ; This Caché command is sent (unsuccessfully) to CacheEvalA
 ; since it is preceded with a "=".
 : =Write X<SYNTAX>
 ; This next entry will end this session with the current
 ; CacheStart parameters.
 : ~
 ; You have the opportunity to reset the
 ; CacheStart parameters and restart.
Restart? Y
 : A 10X is the flag CACHE_TTNONE, which means that there will
 ; be no output printed.
Enter term flags (in hex): 10
Enter timeout: 30
Enter principal output device: <RETURN>
Enter principal input device: <RETURN>
:Write ^=TEMP
 : ~
Restart? No
$
This chapter covers the following topics:

- Managing Transactions Within Applications
- Automatic Transaction RollBack
- System-Wide Issues with Transaction Processing

A transaction is a logical unit of work. Transaction processing helps maintain the logical integrity of your database.

For example, when transferring money from one account to another, a bank may need to subtract an amount from a field in one table and add the same amount to a field in another table. By specifying that both updates form a single transaction, you ensure that either both operations are performed or neither is performed, which means that one cannot be executed without the other.

Within your application, a single SQL INSERT, UPDATE, or DELETE statement, or a single global Set or Kill, may not in itself constitute a complete transaction. In such cases, you use transaction processing commands to define the sequence of operations that forms a complete transaction. One command marks the beginning of the transaction; after a sequence of possibly many commands, another command marks the end of the transaction.

Under normal circumstances, the transaction executes in its entirety. If a program error or system malfunction leads to an incomplete transaction, then the part of the transaction that was completed is rolled back.

Application developers should handle transaction rollback within their applications. Caché also handles transaction rollback automatically in the event of a system failure and at various junctures, such as recovery and during HALT or ResJob.
In Caché, the error trap STU always rolls back a transaction. Caché records rollbacks in the cconsole.log file if there is a \%^LOGROLLBACK\ global.

14.1 Managing Transactions Within Applications

In Caché, you define transactions within applications using either

- SQL statements in macro source routines
- Caché ObjectScript commands

Both methods work, regardless of whether the database modifications that constitute the transactions are performed with SQL INSERT, UPDATE, and DELETE statements or Caché ObjectScript Set and Kill commands.

14.1.1 Transaction Commands

Caché supports the ANSI SQL operations COMMIT WORK and ROLLBACK WORK, as well as the Caché extensions %BEGTRANS and %INTRANS. In addition, Caché implements some of the transaction commands that are part of the M Type A standard.

These SQL and Caché ObjectScript commands are summarized in the following table.
## Caché Transaction Commands

<table>
<thead>
<tr>
<th>SQL Command</th>
<th>ObjectScript Command</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>%BEGTRANS</td>
<td>TSTART</td>
<td>Marks the beginning of a transaction.</td>
</tr>
</tbody>
</table>
| %INTRAN       | N/A                  | Detects whether a transaction is currently in progress:  
|               |                      | • <0 means in a transaction, but journaling disabled.  
|               |                      | • 0 means not in a transaction.  
|               |                      | • >0 means in a transaction.  
|               |                      | $ZU(34) returns the value. |
| COMMIT WORK   | TCOMMIT              | Signals a successful end of transaction. |
| ROLLBACK WORK | TROLLBACK            | Signals an unsuccessful end of transaction; all the database updates performed since the beginning of transaction should be rolled back or undone. |

**Note:** To prevent the Set or Kill commands from being invoked within a transaction from being seen outside the transaction, you must coordinate access to the global(s) via the Lock command.

### 14.1.2 Using the Lock Command in Transactions

Whenever you access a global which might be accessed by more than one user, you need to protect the integrity of the database by using the Lock command on that global.

Unlock is the releasing of an item that your process has locked. The Lock command with no arguments unlocks all your items. For example,

```
Lock ^A
```

unlocks all your items before attempting to lock ^A.

```
Lock -^A
```

unlocks ^A (if you had it locked) without affecting other lock items. The system manager can unlock any item with the LOCKTAB program.
If you unlock a global within a transaction before the transaction is complete, Caché defers the unlock until the transaction is committed or rolled back. Within the transaction, the data appears to be unlocked, permitting a subsequent lock of the same value within the same transaction to work correctly. Outside of the transaction, however, the data remains locked.

Why defer unlocks within transactions? Consider the following common case, which adds a name and related information to the end of a list. ^A(0) contains the last-used number, for instance 4, as you enter this code:

```
##sql(%BEGTRANS)
Lock ^A(0) set id=^A(0)+1,^A(0)=id
Lock ^A(id) set ^A(id)=name // processing continues ...
Lock
##sql(COMMIT WORK)
```

You are sure to get an id equal to 5. The second Lock allows another caller of this code to get an id equal to 6 without interference, while we complete the transaction on id equal to 5. So far, this logic appears to work correctly. If an error occurs, however, and the transaction is rolled back, which resets ^A(0) to 4, a third caller gets an id equal to 5, and a fourth caller gets an id equal to 6.

To prevent this from happening, Caché defers the effect of all unlocks until the transaction is committed or rolled back. After your second Lock in the example, a display of the lock table by LOCKTAB shows your lock on ^A(id) but not on ^A(0). Nevertheless, another process cannot lock ^A(0) until you complete your transaction.

If you do not wish to hold the lock throughout a possibly lengthy transaction, split it into two transactions, as in this example:

```
##sql(%BEGTRANS)
Lock ^A(0) set id=^A(0)+1,^(0)=id
##sql(COMMIT WORK)

##sql(%BEGTRANS)
Lock ^A(id) set ^A(id)=name // processing continues ...
##sql(COMMIT WORK)
Lock
```

If the lock table fills up, Caché sends the following message to the econsole.log file:

```
LOCK TABLE FULL
```

Filling the lock table is not an error condition as far as an application is concerned, unless a deadlock occurs; jobs will just wait until there is space to add their locks to the lock table. However, if all the jobs that have been granted locks at a particular time want to add more locks and no one is willing to release any locks, the application will get a deadlock. This is considered an application programming error.
14.1.2.1 Using $INCREMENT in Transactions

A call to the $INCREMENT function is not considered part of a transaction. It is not rolled back as part of transaction rollback. $INCREMENT can be used to get an index value without using the Lock command. This is advantageous for transactions where you may not want to lock the counter global for the duration of the transaction.

Note: $INCREMENT may be incremented by one process within a transaction and, while that transaction is still processing, be incremented by another process in a parallel transaction. If the first transaction rolls back, there may be a “skipped” increment, “wasting” a number.

14.1.3 Transaction Rollback Within An Application

If you encounter an error during a transaction, you can roll it back in three ways:

• Issue the SQL rollback command, RollBack WORK
• Issue the Caché ObjectScript rollback command, TRollBack
• Make a call to %ETN

14.1.3.1 Issue an SQL or Caché ObjectScript Rollback Command

Application developers can use two types of rollback commands to designate the unsuccessful end of a transaction and automatically roll back incomplete transactions:

• Use ##sql(ROLLBACK WORK), in the macro source routine.
• Use the Caché ObjectScript TRollBack command, in macro or intermediate source code.

The rollback command must cooperate with an error trap, as in the following example:

ROU        ##sql(%BEGTRANS) set $ZT="ERROR"
Set ^ZGLO(1)=100
Set ^ZGLO=error
Set ^ZGLO(1,1)=200
##sql(COMMIT WORK) Write !,"Transaction Committed" Quit
ERROR        ##sql(ROLLBACK WORK)
Write !,"Transaction failed." Quit

In the example code, $ZT is set to run the subroutine ERROR if a program error occurs before the transaction is committed. Line ROU begins the transaction and sets the error trap. Lines ROU+1 and ROU+3 set the nodes of the global ^ZGLO. However, if the variable error is undefined, ROU+2 causes a program error and line ROU+3 does not execute. Program execution goes to the subroutine ERROR and the set of ^ZGLO(1) is undone. If line ROU+2
were deleted, \(^ZGLO\) would have its value set both times, the transaction would be committed, and the message “Transaction committed” would be written.

### 14.1.3.2 Make a Call To \(\%ETN\)

If you have not handled transaction rollback with a rollback command, the error trap utility \(\%ETN\) detects incomplete transactions and prompts the user to either commit or rollback the transaction. You should handle rollback within your application, since committing an incomplete transaction usually leads to degradation of logical database integrity.

If you run \(\%ETN\) after an error when a transaction is in progress, the following rollback prompt is displayed:

You have an open transaction.  
Do you want to perform a Commit or Rollback?  
Rollback =>

If there is no response within a 10-second timeout period, the system defaults to rollback. In a jobbed job or an application mode job the transaction is rolled back with no message.

\(\%ETN\) itself does not do anything to trigger transaction rollback, but it typically ends by halting out of Caché. Transaction rollback occurs when you halt out of Caché ObjectScript and the system runs \(\%HALT\) to perform Caché process cleanup. There is an entry point into \(\%ETN\), called \(\text{BACK}^\%\%ETN\), which ends with a quit, rather than a halt. If a routine calls \(\text{BACK}^\%\%ETN\), rather than \(^\%\%ETN\) or \(\text{FORE}^\%\%ETN\), it will not perform transaction rollback as part of the error handling process.

### 14.1.4 Examples of Transaction Processing Within Applications

The following example shows how transactions are handled in macro source routines. It performs database modifications with SQL code. The SQL statements transfer funds from one account to another:

```objectscript
Transfer(from,to,amount)   // Transfer funds from one account to another
{
    TSTART
    \&SQL(UPDATE A.Account
        WHERE A.Account.AccountNum = :from)
    If SQLCODE TRollBack  Quit "Cannot withdraw, SQLCODE = "_SQLCODE
    \&SQL(UPDATE A.Account
        WHERE A.Account.AccountNum = :to)
    If SQLCODE TROLLBACK  QUIT "Cannot deposit, SQLCODE = "_SQLCODE
    TCOMMIT
    QUIT "Transfer succeeded"
}
```
14.2 Automatic Transaction RollBack

Transaction rollback occurs automatically during:

- Caché startup, if recovery is needed
- **HALT**
- **RESJOB**

In addition, system managers can roll back incomplete transactions in cluster-specific databases by running the Restore Journal option of the **JOURNAL** utility.

14.2.1 Rollback During Caché Startup

When you start Caché and it determines that recovery is needed, any transaction on the computer that was incomplete will be rolled back.

14.2.2 Rollback at Halt from Caché

If you issue **Halt** from Caché when there are transactions in progress within your process, the system prompts you to commit or rollback the transactions. Jobbed jobs and application mode jobs roll back without prompting.

14.2.3 Rollback During RESJOB

14.2.3.1 Jobbed Processes and Application Mode Processes

If you issue a **RESJOB** command on a jobbed process which has transactions in progress, the system automatically rolls back those transactions.

14.2.3.2 Programmer Mode User Processes

If you issue a **RESJOB** command on a programmer mode user process, the system displays a message to the user, asking whether they want the transactions committed or rolled back.

14.2.4 Restore Journal Option of JOURNAL Utility

When you select the Restore Journal option from the **JOURNAL** utility main menu, the journal file is restored and all incomplete transactions are rolled back.
14.3 System-Wide Issues with Transaction Processing

You and your system management personnel need to keep the following system-wide issues in mind when you implement transaction processing:

- Backups
- Journaling

14.3.1 Backups with Transaction Processing

With transaction processing, make sure there are no transactions open when you clear the journal after you complete your backup. To do so:

- On Windows, stop and start Caché, then set switch 10 with the SWSET utility.
- On UNIX, run `mstop` to shut down Caché. Then, run `mstart` and set switch 10 with the SWSET utility.
- On OpenVMS, run `MSTOP` to shut down Caché. Then, run `MSTART` and set switch 10 with the SWSET utility.

Alternatively, you can ensure all users are off the system.

14.3.2 Journaling with Transaction Processing

To use transaction processing, your system manager must enable journaling. With one exception, Caché makes a record in the journal file for:

- All transaction commands
- All `Set` and `Kill` commands that occur within a transaction

This does not apply to globals mapped to the CACHETEMP database — these are not journaled. After journaling is started nothing further need be done. You do not have to say yes to journal all `Set` and `Kill` calls nor do you have to use `%JOURNAL` to specify which globals to journal. All calls to `Set` and `Kill` within a transaction are automatically journaled if journaling is on.
Note: If the system manager stops journaling while a process has an open transaction and you then run `##sql(%INTRANS)` from the process, the `%INTRANS` call does not show that journaling is disabled.

### 14.3.3 Globals to Avoid in Transactions

You should not perform transaction processing with globals on other computers in networked configurations, as Caché does not journal them automatically.

### 14.3.4 Caché Network Transaction Processing

Currently, transaction rollback for database changes made across the network is not supported.

### 14.3.5 Asynchronous Error Notifications

You can specify whether a job can be interrupted by asynchronous errors using by setting `$ZU(68,34)$:

- `$ZU(68,34,1)` enables the reception of asynchronous errors.
- `$ZU(68,34,0)` disables the reception of asynchronous errors.

$ZU(69,34)$ sets a system-wide default for new processes for whether processes are willing to be interrupted by asynchronous errors. It defaults to 1, meaning “YES.”

If multiple asynchronous errors are detected for a particular job, the system triggers at least one such error. However, there is no guarantee which error will be triggered.

The asynchronous errors currently implemented include:

- `<LOCKLOST>` — Some locks once owned by this job have been reset.
- `<DATALOST>` — Some data modifications performed by this job have received an error from the server.
- `<TRANLOST>` — A distributed transaction initiated by this job has been asynchronously rolled back by the server.

Even if you disable a job receiving asynchronous errors, the next time the job performs a `ZSync` command, the asynchronous error is triggered.

At each `TStart`, `TCommit`, or `Lock` operation, and at each network global reference, Caché checks for pending asynchronous errors. Since `Set` and `Kill` operations across the network
are asynchronous, an arbitrary number of other instructions may interpose between when the **Set** is generated and when the asynchronous error is reported.
This chapter discusses Caché error processing. Topics include:

- How Error Processing Works
- Handling Errors with $ETRAP
- Handling Errors with $ZTRAP
- Handling Errors in an Error Handler
- Forcing an Error
- Processing Errors in Programmer Mode

15.1 How Error Processing Works

Managing the behavior of Caché when an error occurs is called *error processing* or *error handling*. You establish an error handler to perform one or more of the following functions:

- Correct the condition that caused the error
- Perform some action that allows execution to resume despite the error
- Divert the flow of execution
- Log information about the error

To do this, Caché provides the functionality so that you can provide your application with an *error handler*. An error handler processes any error that may occur while the application
is running. It is a special variable that defines one or more Caché ObjectScript routines to be executed when an error occurs; it has access to error information that Caché places in a special variable when the error occurs.

To set up an error handler, the basic process is:

1. Create one or more routines to perform error processing. This allows you to perform customized error handling for each particular part of an application.
2. Create one or more error handlers within your application, each pointing to a particular error processing procedure, such as:

   NEW $ETRAP
   SET $ETRAP="GOTO LogError^ErrRoutine"

The above code uses $ETRAP, which is the default Caché error handler. If you have set an error trap in $ETRAP, then whenever an error occurs, Caché passes through the following process:

1. Caché executes the commands specified by $ETRAP.
2. If you signed onto Caché in Programmer Mode and have not set an error trap, Caché displays an error message on the principal device and enters Programmer Mode with the program stack intact. The programmer can later resume execution of the program.
3. If you invoked Caché in Application Mode and have not set an error trap, Caché displays an error message on the principal device and executes a HALT command.

Note: $ETRAP is the default Caché error handler, and InterSystems recommends its use for error processing. There is an alternate error handler, $ZTRAP, the use of which is not recommended, except under special circumstances, when specifically recommended by a representative of the InterSystems Worldwide Response Center.

15.1.1 Internal Error-Trapping Behavior

To get the full benefit of Caché error processing and the scoping issues surrounding the $ETRAP special variable (as well as $ZTRAP), it is helpful to understand how Caché transfers control from one routine to another.

Caché builds a data structure called a “DO frame” or context frame each time any of the following occurs:

- A routine calls another routine with a DO command.
- An XECUTE command argument causes Caché ObjectScript code to execute.
• A user-defined function is executed.

The frame is built on the call stack, one of the private data structures in the address space of your process. Caché stores the following elements in the frame for a routine:

• The value of the $ETRAP special variable (if any)
• The value of the $ZTRAP special variable (if any)
• The position to return from the subroutine

When routine A calls routine B with `DO ^B`, Caché builds a DO frame on the call stack to preserve the context of A. When routine B calls routine C, Caché adds a DO frame to the call stack to preserve the context of B, and so forth.

**Frames on a Call Stack**

If routine A in the figure above is invoked at the Programmer Mode prompt using the DO command, then an extra DO frame, not described in the figure, exists at the base of the call stack.
15.1.2 Current Context Level

You can use the following to return information about the current context level:

- The `$STACK` special variable contains the current stack level.
- The `$ESTACK` special variable contains the current stack level. It can be initialized to 0 (level zero) at any user-specified point.
- The `$STACK` function returns information about the current context and contexts that have been saved on the call stack.

15.1.2.1 The `$STACK` Special Variable

The `$STACK` special variable contains the number of frames currently saved on the call stack for your process. The `$STACK` value is essentially the context level number (zero based) of the currently executing context. Therefore, when a Caché image is started, but before any commands are processed, the value of `$STACK` is 0.

See the `$STACK` special variable in the Caché ObjectScript Reference for details.

15.1.2.2 The `$ESTACK` Special Variable

The `$ESTACK` special variable is similar to the `$STACK` special variable, but is more useful in error handling because you can reset it to 0 (and save its previous value) with the `NEW` command. Thus, a process can reset `$ESTACK` in a particular context to mark it as a `$ESTACK` level 0 context. Later, if an error occurs, error handlers can test the value of `$ESTACK` to unwind the call stack back to that context.

See the `$ESTACK` special variable in the Caché ObjectScript Reference for details.

15.1.2.3 The `$STACK` Function

The `$STACK` function returns information about the current context and contexts that have been saved on the call stack. For each context, the `$STACK` function provides the following information:

- The type of context (DO, XECUTE, or user-defined function)
- The entry reference and command number of the last command processed in the context
- The source routine line or XECUTE string that contains the last command processed in the context
• The $ECODE value of any error that occurred in the context (available only during error processing when $ECODE is non-null)

When an error occurs, all context information is immediately saved on your process error stack. The context information is then accessible by the $STACK function until the value of $ECODE is cleared by an error handler. In other words, while the value of $ECODE is non-null, the $STACK function returns information about a context saved on the error stack rather than an active error handler context at the same specified context level.

See the $STACK function in the Caché ObjectScript Reference for details.

When an error occurs and an error stack already exists, Caché records information about the new error at the context level where the error occurred, unless information about another error already exists at that context level on the error stack. In this case, the information is placed at the next level on the error stack (regardless of the information that may already be recorded there).

Therefore, depending on the context level of the new error, the error stack may extend (one or more context levels added) or information at an existing error stack context level may be overwritten to accommodate information about the new error.

Keep in mind that you clear your process error stack by clearing the $ECODE special variable.

15.1.3 $ETRAP Error Handlers

The $ETRAP special variable can contain one or more Caché ObjectScript commands that are executed when an error occurs. Use the SET command to set $ETRAP to a string that contains one or more Caché commands that transfer control to an error-handling routine. This example transfers control to the LogError procedure (which is part of the routine ErrRoutine):

```
SET $ETRAP="GOTO LogError^ErrRoutine"
```

The commands in the $ETRAP special variable are always followed by an implicit QUIT command. The implicit QUIT command quits with a null string argument when the $ETRAP error handler is invoked in a user-defined function context where a QUIT with arguments is required.

By default, $ETRAP has a scope that is global. Its value is automatically carried forward into new DO, user-defined function, and XECUTE contexts. However, you can explicitly limit the scope of $ETRAP by using the NEW command.
Note: Because $ETRAP is global in scope by default, a routine that is executed in Programmer Mode can define a value for $ETRAP that persists even after the routine terminates. Because the next routine that is executed in Programmer Mode can be affected adversely by a leftover $ETRAP value, Caché invalidates the value of $ETRAP every time it re-displays the Programmer Mode prompt. $ETRAP remains ineligible for use in error handling until it is explicitly set again.

See the $ETRAP special variable in the Caché ObjectScript Reference for details.

15.1.4 Error Codes

When an error occurs, Caché sets the $ZERROR and $ECODE special variables to a value describing the error.

15.1.4.1 $ZERROR Value

Caché sets $ZERROR to a string containing:

- The Caché error code
- The label and routine name where the error occurred

The following examples show the type of messages to which $ZERROR is set when Caché encounters an error. If a variable was undefined in line A+2 of routine ROU, $ZERROR contains:

<UNDEFINED>A+2^ROU

If a syntax error occurs in line TAG of routine ABC, $ZERROR contains:

<SYNTAX>TAG^ABC

You can also explicitly set the special variable $ZERROR as any string up to 128 characters; for example:

SET $ZERROR="Any String"

The $ZERROR special variable retains its value until an error occurs or until you set it to a new value.

See the $ZERROR special variable in the Caché ObjectScript Reference for details.
15.1.4.2 $ECODE Value

When an error occurs, Caché sets $ECODE to the value of a comma-surrounded string containing the ANSI Standard error code that corresponds to the error. For example, when you make a reference to an undefined global variable, Caché sets $ECODE set to the following string:

,,M7,,

If the error has no corresponding ANSI Standard error code, Caché sets $ECODE to the value of a comma-surrounded string containing the Caché error code preceded by the letter Z. For example, if a process has exhausted its symbol table space, Caché places the error code <STORE> in the $ZERROR special variable and sets $ECODE to this string:

,,ZSTORE,,

After an error occurs, your error handlers can test for specific error codes by examining the value of the $ZERROR special variable or the $ECODE special variable.

Note: Error handlers should examine $ZERROR rather than $ECODE special variable for specific errors.

See the $ECODE special variable in the Caché ObjectScript Reference for details.

15.2 Handling Errors with $ETRAP

When an error trap occurs and you have set $ETRAP, Caché takes the following steps:

1. Sets the values of $ECODE and $ZERROR.
2. Processes the commands that are the value of $ETRAP.

By default, each DO, XECUTE, or user-defined function context inherits the $ETRAP error handler of the frame that invoked it. This means that the designated $ETRAP error handler at any context level is the last defined $ETRAP, even if that definition was made several stack levels down from the current level.

See the $ETRAP special variable in the Caché ObjectScript Reference for details.

15.2.1 Context-Specific $ETRAP Error Handlers

Any context can establish its own $ETRAP error handler by taking the following steps:
1. Use the **NEW** command to create a new copy of **$ETRAP**.

2. Set **$ETRAP** to a new value.

If a routine sets **$ETRAP** without first creating a new copy of **$ETRAP**, a new **$ETRAP** error handler is established for the current context, the context that invoked it, and possibly other contexts that have been saved on the call stack. Therefore InterSystems recommends that you create a new copy of **$ETRAP** before you set it.

Keep in mind that creating a new copy of **$ETRAP** does not clear **$ETRAP**. The value of **$ETRAP** remains unchanged by the **NEW** command.

The figure below shows the sequence of **$ETRAP** assignments that create the stack of **$ETRAP** error handlers. As the figure shows:

- Routine A creates a new copy of **$ETRAP**, sets it to “GOTO ^ERR”, and contains the **DO** command to call routine B.
- Routine B does nothing with **$ETRAP** (thereby inheriting the **$ETRAP** error handler of Routine A) and contains the **DO** command to call routine C.
- Routine C creates a new copy of **$ETRAP**, sets it to “GOTO ^CERR”, and contains the **DO** command to call routine D.
- Routine D creates a new copy of **$ETRAP** and then clears it, leaving no **$ETRAP** error handler for its context.

If an error occurs in routine D (a context in which no **$ETRAP** error handler is defined), Caché removes the **DO** frame for routine D from the call stack and transfers control to the **$ETRAP** error handler of Routine C. The **$ETRAP** error handler of Routine C, in turn, dispatches to ^CERR to process the error. If an error occurs in Routine C, Caché transfers control to the **$ETRAP** error handler of Routine C, but does not unwind the stack because the error occurs in a context where a **$ETRAP** error handler is defined.
$ETRAP Error Handlers

15.2.2 $ETRAP Flow of Control Options

When the $ETRAP error handler has been invoked to handle an error and perform any cleanup or error-logging operations, it has the following flow-of-control options:

- Handle the error and continue the application.
- Pass control to another error handler.
- Terminate the application.

15.2.2.1 Handling the Error and Continuing the Application

When a $ETRAP error handler is called to handle an error, Caché considers the error condition active until the error condition is dismissed. You dismiss the error condition by setting the $ECODE special variable to the null string:

```
SET $ECODE=""
```
Clearing $ECODE also clears the error stack for your process.

Typically, you use the GOTO command to transfer control to a predetermined restart or continuation point in your application after the error condition is dismissed. In some cases, you may find it more convenient to quit back to the previous context level after dismissing the error condition.

### 15.2.2.2 Passing Control to Another Error Handler

If the error condition is not dismissed, Caché passes control to another error handler on the call stack when a QUIT command terminates the context at which the $ETRAP error handler was invoked. Therefore, you pass control to a previous level error handler by performing a QUIT from a $ETRAP context without clearing $ECODE.

If routine D, called from routine C, contains an error that transfers control to ^CERR, the QUIT command in ^CERR that is not preceded by setting $ECODE to "" (the empty string) transfers control to the $ETRAP error handler at the previous context level. In contrast, if the error condition is dismissed by clearing $ECODE, a QUIT from ^CERR transfers control to the statement in routine B that follows the DO ^C command.

### 15.2.2.3 Terminating the Application

If no previous level error handlers exist on the call stack and a $ETRAP error handler performs a QUIT without dismissing the error condition, the application is terminated. In Application Mode, Caché is then run down and control is passed to the operating system. In Programmer Mode, the Programmer Mode prompt then appears.

Keep in mind that you use the QUIT command to terminate a $ETRAP error handler context whether or not the error condition is dismissed. Because the same $ETRAP error handler can be invoked at context levels that require an argumentless QUIT and at context levels (user-defined function contexts) that require a QUIT with arguments, the $QUIT special variable is provided to indicate the QUIT command form required at a particular context level.

The $QUIT special variable returns 1 (one) for contexts that require a QUIT with arguments and returns 0 (zero) for contexts that require an argumentless QUIT.

A $ETRAP error handler can use $QUIT to provide for either circumstance as follows:

```
Quit:$QUIT "" Quit
```

When appropriate, a $ETRAP error handler can terminate the application using the HALT command. HALT immediately passes control to the operating system.
15.3 Handling Errors with $ZTRAP

15.3.1 Setting Up $ZTRAP

The $ZTRAP special variable can contain an entry reference that specifies the line or routine, or both, to which control is to be transferred when an error occurs. Use the SET command to give $ZTRAP the value of an entry reference. The following example establishes LOGERR^ERROU as an error-handling routine:

```
SET $ZTRAP="LOGERR^ERROU"
```

$ZTRAP has two basic forms:

- SET $ZTRAP="routinename" which executes in the context in which $ZTRAP was defined. It has a scope that is local to the routine, user-defined function, or XECUTE context in which it was defined.
- SET $ZTRAP="*routinename" which executes in the context in which the error occurred that invoked it.

The system-supplied error routines SET $ZTRAP="^%ET" and SET $ZTRAP="^%ETN" execute in the context in which the error occurred that invoke them.

Note: When $ZTRAP is set to a non-empty value, it takes precedence over any existing $ETRAP error handler. Caché implicitly performs a NEW $ETRAP command and sets $ETRAP equal to "".

15.3.2 Using $ZTRAP

Each routine in an application can establish its own $ZTRAP error handler by setting $ZTRAP. When an error trap occurs, Caché takes the following steps:

1. Sets the special variable $ZERROR to an error message.
2. Resets the program stack to the state it was in when the error trap was set (when the SET $ZTRAP= was executed). In other words, the system removes all entries on the stack until it reaches the point at which the error trap was set. (The program stack is not reset if $ZTRAP was set to a string beginning with an asterisk (*).)
3. Resumes the program at the location specified by the value of $ZTRAP. The value of $ZTRAP remains the same.
Note: You can explicitly set the variable \$ZERROR as any string up to 128 characters. The \$ZERROR special variable retains its value until an error occurs or until you set it to a new value. Usually you would set \$ZERROR only to a null string, but you can set \$ZERROR to a value.

See the \$ZTRAP special variable in the Caché ObjectScript Reference for details.

15.3.3 Unstacking NEW Commands With Error Traps

When an error trap occurs and the program stack entries are removed, Caché also removes all stacked NEW commands back to the subroutine level containing the SET \$ZTRAP=. However, all NEW commands executed at that subroutine level remain, regardless of whether they were added to the stack before or after \$ZTRAP was set.

For example:

```
A
  SET A=1,B=2,C=3,D=4,E=5,F=6
  NEW A,B
  SET \$ZTRAP="ERR"
  NEW C,D
  DO B
  QUIT

B
  NEW E,F
  WRITE 6/0  // Error: division by zero
  QUIT
ERR
  WRITE !,"Error is: ",\$ZERROR
  WRITE
  QUIT
```

When the error in B activates the error trap, the former values of E and F, stacked in line B, are removed, but A, B, C, and D remain stacked.

15.3.4 \$ZTRAP Flow of Control Options

After a \$ZTRAP error handler has been invoked to handle an error and has performed any cleanup or error logging operations, the error handler has three flow control options:

- Handle the error and continue the application.
- Pass control to another error handler
- Terminate the application
15.3.4.1 Continuing the Application

After a $ZTRAP error handler is called to handle an error, you can continue the application. You do not have to clear the value of the $ECODE special variable to continue normal application processing. However, you should clear $ZTRAP to avoid a possible infinite error handling loop if another error occurs. See “Handling Errors in an Error Handler” for more information.

Your error handler should also clear the $ZERROR special variable. After clearing $ZERROR, your $ZTRAP error handler can use the GOTO command to transfer control to a predetermined restart or continuation point in your application to resume normal application processing.

15.3.4.2 Passing Control to Another Error Handler

If the error condition cannot be corrected by a $ZTRAP error handler, you can use the ZQUIT command to transfer control to another error handler. The ZQUIT command re-signals the error condition and causes Caché to unwind the call stack to another call stack level with a $ZTRAP error handler:

- Specify “1” as an argument to ZQUIT to have Caché unwind the call stack to the first encountered $ZTRAP error handler.
- Specify “2” as an argument to ZQUIT to have Caché unwind the call stack to the second encountered $ZTRAP error handler, and so forth.

After Caché has unwound the call stack to the level of the specified error handler, processing continues with the same $ZTRAP error handler that issued the ZQUIT command. That error handler can then issue a GOTO command to transfer control to the $ZTRAP error handler at the new call stack level. For example:

ZQUIT 1 GOTO @$ZTRAP

The following figure shows the flow of control in $ZTRAP error handling routines.
15.4 Handling Errors in an Error Handler

When an error occurs in an error handler, the flow of execution depends on the type of error handler that is currently executing.

15.4.1 Errors in a $ETRAP Error Handler

If the new error occurs in a $ETRAP error handler, Caché unwinds the call stack until the context level at which the $ETRAP error handler was invoked has been removed. Caché then passes control to the next error handler (if any) on the call stack.
15.4.2 Errors in a $ZTRAP Error Handler

If the new error occurs in a $ZTRAP error handler, Caché passes control to the first error handler it encounters, unwinding the call stack only if necessary. Therefore, if the $ZTRAP error does not clear $ZTRAP at the current stack level and another error subsequently occurs in the error handler, the $ZTRAP handler is invoked again at the same context level, possibly causing an infinite error-handling loop.

15.4.3 Error Information in the $ZERROR and $ECODE Special Variables

If another error occurs during the handling of the original error, information about the second error replaces the information about the original error in the $ZERROR special variable. However, Caché appends the new information to the $ECODE special variable. Depending on the context level of the second error, Caché may append the new information to the process error stack as well.

If the existing value of the $ECODE special variable is non-null, Caché appends the code for the new error to the current $ECODE value as a new comma piece. Error codes accrue in the $ECODE special variable until either of the following occurs:

- You explicitly clear $ECODE, for example:

  ```objectscript
  SET $ECODE = ""
  ```

- The length of $ECODE exceeds the maximum string length.

Then, the next new error code replaces the current list of error codes in $ECODE.

When an error occurs and an error stack already exists, Caché records information about the new error at the context level where the error occurred, unless information about another error already exists at that context level on the error stack. In this case, the information is placed at the next level on the error stack (regardless of the information that may already be recorded there).

Therefore, depending on the context level of the new error, the error stack may extend (one or more context levels added) or information at an existing error stack context level may be overwritten to accommodate information about the new error.

Keep in mind that you clear your process error stack by clearing the $ECODE special variable. See the $ECODE and $ZERROR special variables in the Caché ObjectScript Reference for details.
15.5 Forcing an Error

You set the $ECODE special variable or use the ZTRAP command to cause an error to occur under controlled circumstances.

15.5.1 Setting $ECODE

You can set the $ECODE special variable to any non-null string to cause an error to occur. When your routine sets $ECODE to a non-null string, Caché sets $ECODE to the specified string and then generates an error condition. The $ZERROR special variable in this circumstance is set with the following error text:

<EODETRAP>

Control then passes to error handlers as it does for normal application-level errors.

You can add logic to your error handlers to check for errors caused by setting $ECODE. Your error handler can check $ZERROR for an <EODETRAP> error (for example, "$ZE["EODETRAP"]") or your error handler can check $ECODE for a particular string value that you choose.

15.5.2 Creating Application-Specific Errors

Keep in mind that the ANSI Standard format for $ECODE is a comma-surrounded list of one or more error codes:

- Errors prefixed with “Z” are implementation-specific errors
- Errors prefixed with “U” are application-specific errors

You can create your own error codes following the ANSI Standard by having the error handler set $ECODE to the appropriate error message prefixed with a “U”.

SET $ECODE=",Upassword expired,"

15.6 Processing Errors in Programmer Mode

When you generate an error after you sign onto Caché in Programmer Mode with no error handler set, Caché takes the following steps when an error occurs in a line of code you enter:
1. Caché displays an error message on the process' principal device
2. Caché breaks at the call stack level where the error occurred
3. The process enters Programmer Mode.

This same sequence takes place when an **XECUTE** statement originating in Programmer Mode causes an error.

### 15.6.1 Understanding Error Message Formats

As an error message, Caché displays:

1. The entire line of source code in which the error occurred.
2. A line in which a caret (^) points to the particular command that caused the error.
3. A line containing the contents of `$ZERROR`.

In this example, the **SET** command in line TAG+3 of routine ROU has a syntax error:

```objectscript
SET A = B, C = D*2, F = D'+3 DO X
^<SYNTAX>TAG+3^ROU
```

### 15.6.2 Understanding the Programmer Mode Prompt

If the error takes place in a line of Programmer-Mode code, Caché displays the Programmer-Mode prompt after printing the error message, as in this example where `%SYS` is the current namespace:

```objectscript
%SYS>
```

However, if an error occurs during routine execution instead of in a line in Programmer Mode, Caché saves the program stack entering Programmer Mode. An extended prompt message appears, such as:

```objectscript
4d0>
```

In this example, the message indicates that there are four entries on the program stack, the last of which is an invoking of **DO**.

### 15.6.3 Recovering from the Error

You can then take any of the following steps:
Error Processing

- Issue commands in Programmer Mode
- View and modify your variables and global data
- Edit the routine containing the error or any other routine
- Execute other routines

Any of these steps can even cause additional errors.

After you have taken these steps, your most likely course is to either resume execution or to delete all or part of the program stack.

15.6.3.1 Resuming Execution at the Next Sequential Command

You can resume execution at the next command after the command that caused the error, simply by entering an argumentless `GOTO` in Programmer Mode:

4d0>GOTO

15.6.3.2 Resuming Execution at Another Line

You can resume execution at another line by issuing a `GOTO` with an argument in Programmer Mode:

4d0>GOTO TAG

15.6.3.3 Deleting the Program Stack

4d0>QUIT
%SYS>

15.6.3.4 Deleting Part of the Program Stack

You can issue a `QUIT` with arguments to delete the last (or last several) invocations of `DO`, `FOR`, `XECUTE`, or user-defined functions:

9f0>QUIT 5
4d0>QUIT 1
3d0>

15.6.4 `<EDITED>` Errors

Any of several conditions can cause an `<EDITED>` error:

- Whenever you execute a `DO` from one routine to another and someone else edits and uses `ZSAVE` to save the first routine before you return to it, you get the error message `<EDITED>` when you try to return to that routine.
• If you enter Programmer Mode after an error or **BREAK** and then edit a line containing a **DO** command that is stacked on the program stack, an **<EDITED>** error occurs when you try to return to that line. This error also occurs under these conditions if you try to edit the line that contains the error or **BREAK** and you attempt to resume execution at the next instruction.

• If you open a file, edit a routine that issues a **USE** command on that file, and then issue the **USE** command on the opened file for the first time, this action generates an **<EDITED>** error.

Frequently when you get an error on a line and enter Programmer Mode, you can simply edit the line, enter the rest of the commands on that line in Programmer Mode, and then use a **GOTO** with a TAG+offset to resume execution at the next line of the routine.

When an **<EDITED>** error occurs, you must use a **QUIT** command in Programmer Mode, with or without arguments, to remove the stacked commands as needed.
16

Command-Line Routine Debugging

This chapter describes the Caché techniques for testing and debugging Caché applications. Its topics include:

• Debugging with the Caché Debugger
• Debugging With Break
• Using %STACK to Display the Stack
• Error Trap Utilities

An important part of application development is routine debugging: the testing and correcting of program code. Caché gives you two ways to debug your routines:

• Use the Break command in routine code to suspend execution and allow you examine what is happening.
• Use the ZBreak command to invoke the Caché Debugger to interrupt execution and allow you to examine both code and variables.

16.1 Debugging with the Caché Debugger

The Caché Debugger lets you test routines by inserting debugging commands directly into your routine code. Then, when you run the code, you can issue commands to test the conditions and the flow of processing within your application. Its major capabilities are:

• Set breakpoints with the ZBreak command at code locations and take specified actions when those points are reached.
• Set watchpoints on local variables and take specified actions when the values of those variables change.

• Interact with Caché during a breakpoint/watchpoint in a separate window.

• Trace execution and output a trace record (to a terminal or other device) whenever the path of execution changes.

• Display the execution stack.

• Run an application on one device while debugging I/O goes to a second device. This enables full screen Caché applications to be debugged without disturbing the application's terminal I/O.

16.1.1 Entering the Debugger from the Control Panel

You can enter the Caché Debugger from the Caché Control Panel, as follows:

1. Start your application running in the background.
2. From the Caché Cube, bring up the Caché Control Panel.
3. Click the Processes task folder to list the current processes.
4. Right-click on the process identifying your application.
5. Click Debug from the shortcut menu.

Caché opens the debugger in a new window, and places you inside the application. All of the Caché Debugger commands are functional.

16.1.2 Using Breakpoints and Watchpoints

The Caché Debugger provides two ways to interrupt program execution:

• Breakpoints
• Watchpoints

A breakpoint is a location in a Caché routine that you specify with the ZBreak command. When routine execution reaches that line, Caché suspends execution of the routine and, optionally, executes debugging actions you define. You can set breakpoints in up to 20 routines. You can set a maximum of 20 breakpoints within a particular routine.

A watchpoint is a variable you identify in a ZBreak command. When its value is changed with a SET or KILL command, you can cause the interruption of routine execution and/or
the execution of debugging actions you define within the **ZBreak** command. You can set a maximum of 20 watchpoints.

Breakpoints and watchpoints you define are not maintained from one session to another. Therefore, you may find it useful to store breakpoint/watchpoint definitions in a routine or **XECUTE** string so it is easy to reinstate them between sessions.

### 16.1.3 Establishing Breakpoints and Watchpoints

You use the **ZBreak** command to establish breakpoints and watchpoints.

#### 16.1.3.1 Syntax

```
ZBreak location[:action:condition:execute_code]
```

where:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>location</strong></td>
<td>Required. Specifies a code location (that sets a breakpoint) or local or system variable (which sets a watchpoint). If the location specified already has a breakpoint/watchpoint defined, the new specification completely replaces the old one.</td>
</tr>
<tr>
<td><strong>action</strong></td>
<td>Optional. Specifies the action to take when the breakpoint/watchpoint is triggered. For breakpoints, the action occurs before the line of code is executed. For watchpoints, the action occurs after the command that modifies the local variable. Actions may be upper- or lower-case, but must be enclosed in quotation marks.</td>
</tr>
<tr>
<td><strong>condition</strong></td>
<td>Specifies an expression that will be evaluated when the breakpoint/watchpoint is triggered. The expression must be surrounded by quotation marks. If the condition is false, the action will not be carried out and the execute_code will not be executed. If a condition is not specified, the default is true.</td>
</tr>
<tr>
<td><strong>execute_code</strong></td>
<td>Specifies Caché ObjectScript code to be executed if the condition is true. The code must be surrounded by quotation marks if it is a literal. This code is executed prior to the action being carried out. Before the code is executed, the value of $TEST is saved. After the code has executed, the value of $TEST as it existed in the program being debugged is restored.</td>
</tr>
</tbody>
</table>

**Note:** Using **ZBreak** with a ? (question mark) displays help.
16.1.3.2 Setting Breakpoints with Code Locations

You specify code locations as a routine line reference that you can use in a call to the $TEXT function. A breakpoint occurs whenever execution reaches this point in the code, before the execution of the line of code. If you do not specify a routine name, Caché assumes the reference is to the current routine.

16.1.3.3 Argumentless GOTO in Breakpoint Execution Code

An argumentless GOTO is allowed in breakpoint execution code. Its effect is equivalent to executing an argumentless GOTO at the debugger Break prompt and execution proceeds until the next breakpoint.

For example, if the routine you are testing is in the current namespace, you can enter location values such as these:

<table>
<thead>
<tr>
<th>Location Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag^rou</td>
<td>Break before the line at tag in the routine rou.</td>
</tr>
<tr>
<td>tag+3^rou</td>
<td>Break before the third line after tag in routine rou.</td>
</tr>
<tr>
<td>+3^rou</td>
<td>Break before the third line in routine rou.</td>
</tr>
</tbody>
</table>

If the routine you are testing is currently loaded in memory (that is, an implicit or explicit ZLOAD was performed), you can use location values such as these:

<table>
<thead>
<tr>
<th>Location Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Break before the line at tag.</td>
</tr>
<tr>
<td>tag+3</td>
<td>Break before the third line after tag.</td>
</tr>
<tr>
<td>+3</td>
<td>Break before the third line.</td>
</tr>
</tbody>
</table>

16.1.3.4 Setting Watchpoints with Local and System Variable Names

Local variable names cause a watchpoint to occur in these situations:

- When the local variable is created
- When a SET statement changes the value of the local variable
- When a KILL statement deletes the local variable

Variable names are preceded by an asterisk, as in *A.

If you specify an array-variable name, the Caché Debugger watches all descendant nodes. For instance, if you establish a watchpoint for array A, a change to A(5) or A(5,1) triggers the watchpoint.
The variable need not exist when you establish the watchpoint.

You can also use the following system variables:

<table>
<thead>
<tr>
<th>System Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ZERROR</td>
<td>Triggered whenever an error occurs, before invoking the error trap.</td>
</tr>
<tr>
<td>$ZTRAP</td>
<td>Triggered whenever an error trap is set or cleared.</td>
</tr>
<tr>
<td>$IO</td>
<td>Triggered whenever explicitly SET.</td>
</tr>
</tbody>
</table>

### 16.1.3.5 Action Argument Values

The following table describes the values you can use for the **ZBreak** action argument.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;B&quot;</td>
<td>Default, except if you include the &quot;T&quot; action, then you must also explicitly include the &quot;B&quot; action, as in ZB &quot;a:&quot;TB&quot;, to actually cause a break. Suspends execution and displays the line at which the break occurred along with a caret (^) indicating the point in the line. Then displays the Programmer prompt and allows interaction. Execution resumes with an argumentless GOTO command.</td>
</tr>
<tr>
<td>&quot;L&quot;</td>
<td>Same as &quot;B&quot;, except GOTO initiates single-step execution, stopping at the beginning of each line. When a DO command, user-defined function, or XECUTE command is encountered, single-step mode is suspended until that command or function completes.</td>
</tr>
<tr>
<td>&quot;L+&quot;</td>
<td>Same as &quot;B&quot;, except GOTO initiates single-step execution, stopping at the beginning of each command. DO commands, user-defined functions, and XECUTE commands do not suspend single-step mode.</td>
</tr>
<tr>
<td>&quot;S&quot;</td>
<td>Same as &quot;B&quot;, except GOTO initiates single-step execution, stopping at the beginning of each command. When a DO command, user-defined function, FOR command, or XECUTE command is encountered, single-step mode is suspended until that command or function completes.</td>
</tr>
<tr>
<td>&quot;S+&quot;</td>
<td>Same as &quot;B&quot;, except GOTO initiates single-step execution, stopping at the beginning of each command. DO commands, user-defined functions, FOR commands, and XECUTE commands do not suspend single-step mode.</td>
</tr>
</tbody>
</table>
**16.1.3.6 ZBreak Examples**

The following example establishes a watchpoint that suspends execution whenever the local variable A is killed. No action is specified, so "B" is assumed.

```
ZBreak *A::"'$D(A)"
```

The following example illustrates the above watchpoint acting on a direct mode Caché command (rather than on a command issued from within a routine). The caret (^) points to the command that caused execution to be suspended:

```
%SYS>K A
K A
^<BREAK>
%SYS Is0>
```

The following example establishes a breakpoint that suspends execution and sets single-step mode at the beginning of the line TAG^ROU.

```
ZBreak TAG^ROU:"L"
```

The following example shows how the break would appear when the routine is run. The caret (^) indicates where execution was suspended.

```
%SYS>DO ^ROU
TAG SET X=1
^<BREAK>TAG^ROU
%SYS 2d0>
```
In the following example, a breakpoint at line TAG^ROU does not suspend execution, because of the "N" action. However, if X<1 when the line TAG^ROU is reached, then FLAG is SET to X.

ZBreak TAG^ROU:"N":"X<1":"S FLAG=X"

The following example establishes a watchpoint that executes the code in ^GLO whenever the value of A changes. The double colon indicates no condition argument.

ZBreak *A:"N":"X ^GLO"

The following example establishes a watchpoint that causes a trace message to display whenever the value of B changes. The trace message will display only if trace mode has been turned on with the ZBreak /TRACE:ON command.

ZBreak *B:"T"

The following example establishes a watchpoint that suspends execution in single-step mode when variable a is set to 5.

ZBreak *a:"S":"a=5"

When the break occurs in the following example, a caret (^) symbol points to the command that caused the variable a to be set to 5.

%SYS>DO ^test
FOR I=1:1:6 S a=a+1
^<BREAK>
test+3^test
%SYS 3f0> W a
5

### 16.1.4 Disabling Breakpoints and Watchpoints

You can disable either:

- Specific breakpoints and watchpoints
- All breakpoints or watchpoints

#### 16.1.4.1 Disabling Specific Breakpoints and Watchpoints

You can disable a breakpoint or watchpoint by preceding the location with a minus sign. The following command disables a breakpoint previously specified for location TAG^ROU:

ZBreak -TAG^ROU
A disabled breakpoint is “turned off”, but Caché remembers its definition. You can enable the disabled breakpoint by preceding the location with a plus sign. The following command enables the previously disabled breakpoint:

```
ZBreak +TAG^ROU
```

### 16.1.4.2 Disabling All Breakpoints and Watchpoints

You can disable all breakpoints or watchpoints by using the plus or minus signs without a location:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZBreak -</td>
<td>Disable all defined breakpoints and watchpoints.</td>
</tr>
<tr>
<td>ZBreak +</td>
<td>Enable all defined breakpoints and watchpoint.</td>
</tr>
</tbody>
</table>

### 16.1.5 Delaying Execution of Breakpoints and Watchpoints

You can also delay the execution of a break/watchpoint for a specified number of iterations. You might have a line of code that appears within a loop that you want to break on periodically, rather than every time it is executed. To do so, establish the breakpoint as you would normally, then disable with a count following the location argument.

The following `ZBreak` command causes the breakpoint at TAG^ROU to be disabled for 100 iterations. On the 101st time this line is executed, the specified breakpoint action occurs.

```
ZBREAK TAB^ROUTINE       ; establish the breakpoint
ZBREAK -TAB^ROUTINE#100  ; disable it for 100 iterations
```

**Important:** A delayed breakpoint is not decremented when a line is repeatedly executed because it contains a `FOR` command.

### 16.1.6 Deleting Breakpoints and Watchpoints

You can delete individual break/watchpoints by preceding the location with a double minus sign; for example:

```
ZBreak --TAG^ROU
```

After you have deleted a breakpoint/watchpoint, you can only reset it by defining it again.

To delete all breakpoints, issue the command:

```
ZBreak /CLEAR
```
This command is performed automatically when a Caché process halts.

### 16.1.7 Single-Step Breakpoint Actions

You can use single step execution to stop execution at the beginning of each line or of each command in your code. You can establish a single step breakpoint to specify actions and execution code to be executed at each step. Use the following syntax to define a single step breakpoint:

```
ZBreak $:action[:condition:execute_code]
```

Unlike other breakpoints, `ZBreak $` does not cause a break, because breaks occur automatically as you single-step. `ZBreak $` lets you specify actions and execute code at each point where the debugger breaks as you step through the routine. It is especially useful in tracing executed lines or commands. For example, to trace executed lines in the application ^TEST:

```
%SYS>ZBreak /TRACE:ON
%SYS>Break "L+"
%SYS>ZBreak $:"T"
```

The “T” action specified alone suppresses the single step break that normally occurs automatically. The “N” action code also suppresses the single step break that normally occurs. Establish the following single step breakpoint definition if both tracing and breaking should occur:

```
%SYS>ZBreak $:"TB"
```

### 16.1.8 Tracing Execution

You can control whether or not the “T” action of the `ZBreak` command is enabled by using the following form of `ZBreak`:

```
ZBreak /TRACE:state[:device]
```

where `state` can be:

<table>
<thead>
<tr>
<th>ON</th>
<th>Enables tracing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>Disables tracing.</td>
</tr>
<tr>
<td>ALL</td>
<td>Enables tracing of application by performing the equivalent of: ZBreak /TRACE:ON[:device] Break &quot;L+&quot; ZBreak $:&quot;T&quot;</td>
</tr>
</tbody>
</table>

---

Using Caché ObjectScript 213
When `device` is used with the ALL or ON state keywords, trace messages are redirected to the specified device rather than to the principal device. If the device is not already open, Caché attempts to open it as a sequential file with WRITE and APPEND options.

When device is specified with the OFF state keyword, Caché closes the file if it is currently open.

**Note:** `ZBreak /TRACE:OFF` does not delete or disable the single-step breakpoint definition set up by `ZBreak /TRACE:ALL`, nor does it clear the “L+” single stepping set up by `ZBreak /TRACE:ALL`. You must also issue the commands `ZBreak --$` and `Break "C"` to remove the single stepping; alternatively, you can use the single command `Break "OFF"` to turn off all debugging for the process.

Tracing messages are generated at breakpoints associated with a “T” action. With one exception, the trace message format is as follows for all breakpoints:

Trace: `ZBreak at line_reference`

where `line_reference` is the line reference of the breakpoint.

The trace message format is slightly different for single step breakpoints when stepping is done by command:

Trace: `ZBreak at line_reference source_offset`

where `line_reference` is the line reference of the breakpoint and `source_offset` is the 0-based offset to the location in the source line where the break has occurred.

Operating System Notes:

- **Windows** — Trace messages to another device are supported on Windows platforms only for terminal devices connected to a COM port, such as COM1:. You cannot use the console or a terminal window. You can specify a sequential file for the trace device

- **UNIX** — To send trace messages to another device on UNIX platforms:
  1. Log in to `/dev/tty01`.
  2. Verify the device name by entering the `tty` command:

     ```bash
     $ tty /dev/tty01
     ```

  3. Issue the following command to avoid contention for the device:

     ```bash
     $ exec sleep 50000
     ```

  4. Return to your working window.
5. Start and enter Caché.

6. Issue your trace command:

   ZB /T:ON:"/dev/tty01"

7. Run your program.

   If you have set breakpoints or watchpoints with the “T” action, you see trace messages appear in the window connected to /dev/tty01.

- **OpenVMS** — To send trace messages to another device on OpenVMS platforms:

  1. Log in on TTA1:

  2. Verify the device name by entering the following command at the DCL prompt:

     \$ WRITE sys\$output f\$getjpi("","TERMINAL")

     TTA1:

  3. Issue the **PROTECTION** command so you have write privileges to the terminal:

     \$SET PROT=W:rwlp TTA1:

  4. Issue the following command to avoid contention for the device:

     \$WAIT 1

  5. Return to your working window or to a terminal where you are logged in on your principal device.

  6. Issue the following command to set your process privileges to share:

     \$SET PROC/PRIV=share

  7. Start Caché.

  8. Issue your trace command:

     ZB /T:ON:"TTA1:"

  9. Run your program.

   If you have set breakpoints or watchpoints with the “T” action, you will see trace messages appear on the window connected to TTA1:

---

### 16.1.8.1 Trace Message Format

If you set a code breakpoint, the following message appears:
If you set a variable watchpoint, one of the following messages appears:

Trace: ZBreak SET var=val at tag2^rou2
Trace: ZBreak SET var=Array Val at tag2^rou2
Trace: ZBreak KILL var at tag2^rou2

- \textit{var} is the variable being watched
- \textit{val} is the new value being set for that variable.

If you issue a \textbf{NEW} command, you receive no trace message. However, the trace on the variable is triggered the next time you issue a \textbf{SET} or \textbf{KILL} on the variable at the NEW level. If a variable is passed by reference to a routine, then that variable is still traced, even though the name has effectively changed.

### 16.1.9 INTERRUPT Keypress and Break

Normally, pressing the interrupt key sequence (typically \texttt{CTRL-C}) generates a trappable (\texttt{<INTERRUPT>}) error. To set interrupt processing to cause a break instead of an \texttt{<INTERRUPT>} error, use the following \texttt{ZBreak} command: \texttt{ZBreak /INTERRUPT:Break}

This causes a break to occur when you press the INTERRUPT key even if you have disabled interrupts at the application level for the device.

If you press the INTERRUPT key during a read from the terminal, you may have to press \texttt{RETURN} to display the break-mode prompt. To reset interrupt processing to generate an error rather than cause a break, issue the following command: \texttt{ZBreak /INTERRUPT:NORMAL}

### 16.1.10 Displaying Information About the Current Debug Environment

To display information about the current debug environment, including all currently defined break or watchpoints, issue the \texttt{ZBreak} command with no arguments.

The argumentless \texttt{ZBreak} command describes the following aspects of the debug environment:

- Whether \texttt{CTRL-C} causes a break
- Whether trace output specified with the “\texttt{T}” action in the \texttt{ZBreak} command displays
- The location of all defined breakpoints, with flags describing their enabled/disabled status, action, condition and executable code
• All variables for which there are watchpoints, with flags describing their enabled/disabled status, action, condition and executable code

Output from this command is displayed on the device you have defined as your debug device, which is your principal device unless you have defined the debug device differently with the ZBreak /DEBUG command described in the Using the Debug Device section.

The following table describes the flags provided for each breakpoint and watchpoint:

<table>
<thead>
<tr>
<th>Display Section</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification of break/watch point</td>
<td>Line in routine for breakpoint. Local variable for watchpoint.</td>
</tr>
<tr>
<td>F:</td>
<td>Flag providing information about the type of action defined in the ZBreak command.</td>
</tr>
<tr>
<td>S:</td>
<td>Number of iterations to delay execution of a breakpoint/watchpoint defined in a ZBreak command.</td>
</tr>
<tr>
<td>C:</td>
<td>Condition argument set in ZBreak command.</td>
</tr>
<tr>
<td>E:</td>
<td>Execute_code argument set in ZBreak command.</td>
</tr>
</tbody>
</table>

The following table describes how to interpret the F: value in a breakpoint/watchpoint display. The F: value is a list of the applicable values in the first column.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Breakpoint or watchpoint enabled</td>
</tr>
<tr>
<td>D</td>
<td>Breakpoint or watchpoint disabled</td>
</tr>
<tr>
<td>B</td>
<td>Perform a break</td>
</tr>
<tr>
<td>L</td>
<td>Perform an &quot;L&quot;</td>
</tr>
<tr>
<td>L+</td>
<td>Perform an &quot;L+&quot;</td>
</tr>
<tr>
<td>S</td>
<td>Perform an &quot;S&quot;</td>
</tr>
<tr>
<td>S+</td>
<td>Perform an &quot;S+&quot;</td>
</tr>
<tr>
<td>T</td>
<td>Output a Trace Message</td>
</tr>
</tbody>
</table>
16.1.10.1 Default Display

When you first enter Caché and use ZB, the output is as follows:

```bash
%SYS>ZB
Break:
No breakpoints
No watchpoints
```

This means:

- Trace execution is OFF
- There is no break if CTRL-C is pressed
- No break/watchpoints are defined

16.1.10.2 Display When Breakpoints and Watchpoints Exist

This example shows two breakpoints and one watchpoint being defined:

```bash
%SYS>ZB +3^test:::"WRITE ""IN test""
%SYS>ZB -+3^test#5
%SYS>ZB +5^test:"L"
%SYS>ZB -+5^test
%SYS>ZB *a:"T":"a=5"
%SYS>ZB /TRACE:ON
%SYS>ZB
Break: TRACE ON
+3^test F:EB S:5 C: E:"WRITE ""IN test""
+5^test F:DL S:0 C: E:
a F:ET S:0 C:"a=5" E:
```

The first two ZBreak commands define a delayed breakpoint; the second two ZBreak commands define a disabled breakpoint; the fifth ZBreak command defines a watchpoint. The sixth ZBreak command enables trace execution. The final ZBreak command, with no arguments, displays information about current debug settings.

In the example, the ZBreak display shows that:

- Tracing is ON
- There is no break if CTRL-C is pressed.

The output then describes the two breakpoints and one watchpoint:

- The F flag for the first breakpoint equals “EB” and the S flag equals 5, which means that a breakpoint will occur the fifth time the line is encountered. The E flag displays executable code, which will run before the Caché Programmer prompt for the break is displayed.
• The F flag for the second breakpoint equals “DL”, which means it is disabled, but if enabled will break and then single-step through each line of code following the breakpoint location.

• The F flag for the watchpoint is “ET”, which means the watchpoint is enabled. Since trace execution is ON, trace messages will appear on the trace device. Because no trace device was defined, the trace device will be the principal device.

• The C flag means that trace is displayed only when the condition is true.

16.1.11 Using the Debug Device

The debug device is the device where:

• The **ZBreak** command displays information about the debug environment.

• The Caché Programmer prompt appears if a break occurs.

**Note:** On Windows platforms, trace messages to another device are supported only for terminal devices connected to a COM port, such as COM1:

When you enter Caché, the debug device will automatically be set to your principal device. At any time, debugging I/O can be sent to an alternate device with the command: **ZB /DEBUG:“device”**.

**Note:** There are also operating-system–specific actions that you can take.

On OpenVMS systems, to cause the break to occur in an X window linked to the device TTA1:, issue the following command:

```
ZB /D:"TTA1:"
```

On UNIX systems, to cause the break to occur on the tty01 device, issue the following command:

```
ZB /D:" /dev/tty01/"
```

When a break occurs, because of a **CTRL-C** or to a breakpoint or watchpoint being triggered, it appears in the window connected to the device. That window becomes the active window. If the device is not already open, an automatic OPEN is performed. If the device is already open, any existing OPEN parameters are respected.
Important: If the device you specify is not an interactive device (such as a terminal), you may not be able to return from a break. However, the system does not enforce this restriction.

16.1.12 Caché Debugger Example

First, suppose you are debugging the simple program named test shown below. The goal is to put 1 in variable a, 2 in variable b and 3 in variable c.

test; Assign the values 1, 2, and 3 to the variables a, b, and c
S a=1
S b=2
S c=3 K a WRITE "in test, at end"
QUIT

However, when you run test, only variables b and c hold the correct values:

%SYS>DO ^test
in test, at end
%SYS>W
b=2
c=3
%SYS>

The problem in the program is obvious: variable a is KILLED on line 4. However, assume you need to use the debugger to determine this.

You can use the **ZBreak** command to set single-stepping through each line of code ("L" action) in the routine test. By a combination of stepping and writing the value of a, you determine that the problem lies in line 4:
%SYS> NEW
%SYS 1S1> ZB
Break
No breakpoints
No watchpoints
%SYS 1S1>ZB "test:"L"
%SYS 1S1> DO ^test
S a=1
^<BREAK>test+1^test
%SYS 3d3> W a
<UNDEFINED>^test
%SYS 3d3> G
S b=2
^<BREAK>test+2^test
%SYS 3d3> W a
1
%SYS 3d3> G
S c=3 K a WRITE "in test, at end"
^<BREAK>test+3^test
%SYS 3d3> W a
1
%SYS 3d3> G
in test, at end
QUIT
^<BREAK>test+4^test
%SYS 3d3> W a
W a
^<UNDEFINED>^test
%SYS 3d3> G
%SYS 1S1>

You can now examine that line and notice the **KILL a** command. In more complex code, you might now want to single-step by command ("S" action) through that line.

If the problem occurred within a DO, FOR or XECUTE command or user-defined function, you would use the "L+" or "S+" actions to single-step through lines or commands within the lower level of code.

### 16.1.13 Understanding Caché Debugger Errors

The Caché Debugger flags an error in a condition or execute argument with an appropriate Caché error message.

If the error is in the execute code parameter, the condition surrounds the execute code when the execute code is displayed prior to the error message. The condition (TEST) is always set back to one at the end of the execution code so that the rest of the debugger processing code works properly. When control returns to the routine, the value of TEST within the routine is restored.

Suppose you issue the following **ZBreak** command for the example program test:
%SYS>ZBreak test+1^test:"B":"a=5":"WRITE b"

In the program test, variable b is not defined at line test+1, so there is an error. The error display appears as follows:

```
i a=5 X "WRITE b" i l
^<UNDEFINED>test+1^test
```

If you had not defined a condition, then an artificial true condition would be defined prior to and after the execution code; for example:

```
%SYS>i 1 WRITE b i l
```

# 16.2 Debugging With Break

Caché includes three forms of the `Break` command:

- The `Break` command without an argument inserted into routine code to suspend a routine and returns a job to Programmer Mode.
- The `Break` command with an argument of 1 or 0 to enable and disable interrupts (`CTRL-C`) from terminals.
- The `Break` command with special debugging arguments to suspend a Caché routine at a designated location so you can later resume execution at the same (or another) location.

When you sign on in Programmer Mode, your process begins with an implicit Break 1 (interrupt enabled) by default. `Break` functionality is not available in Application Mode.

## 16.2.1 Using Argumentless Break to Suspend Routine Execution

To suspend a running routine and return the process to Programmer Mode, enter an argumentless `Break` into your routine at points where you want execution to temporarily stop. (Caché accepts only the abbreviation `B` for the `Break` command.)

When Caché encounters a Break, it takes the following steps:

1. Suspends the running routine
2. Returns the process to Programmer Mode

You can now issue Caché ObjectScript commands, modify data, and execute further routines or subroutines, even those with errors or additional Breaks.
To resume execution at the point at which the routine was suspended, issue an argumentless GOTO command.

### 16.2.2 Using Argumentless Break with a Condition

You may find it useful to specify a condition on an argumentless Break command in code so that you can rerun the same code simply by setting a variable rather than having to change the routine. For example, you may have the following line in a routine:

```ObjectScript
CHECK B: $D (DEBUG)
```

You can then set the variable DEBUG to suspend the routine and return the job to Programmer Mode or clear the variable DEBUG to continue running the routine.

### 16.2.3 Using Argumented Break to Enable or Disable Interrupts

You can use Break with an argument of 1 or 0 to enable or disable interrupts (CTRL-C) from the terminal. Entering B 1 at the Programmer prompt or including it in source code enables user interrupts with a CTRL-C. Entering:

```
B 0
```

at the Programmer prompt or including it in source code disables user interrupts with a CTRL-C.

### 16.2.4 Using Argumented Break to Suspend Routine Execution

You do not have to place argumentless Break commands at every location where you want to suspend your routine. Caché provides several argument variations of the Break command that can periodically suspend a routine as if argumentless Breaks are scattered throughout the code. Variations of the Break command arguments are listed in the following table:
## Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break &quot;S&quot;</td>
<td>Use <code>Break &quot;S&quot;</code> (Single Step) to step through your code a single command at a time, breaking on every Caché ObjectScript command. The system stops breaking when a <strong>Do</strong> command, an <strong>Xecute</strong> command, a <strong>For</strong> loop, or a user-defined function is encountered, and resumes single-step breaking when the command, function or loop is done.</td>
</tr>
<tr>
<td>Break &quot;S+&quot;</td>
<td>Acts like the <code>Break &quot;S&quot;</code> variation except that Caché continues to break on every command when a <strong>Do</strong> command, <strong>Xecute</strong> command, <strong>For</strong> loop, or user-defined function is encountered.</td>
</tr>
<tr>
<td>Break &quot;S-&quot;</td>
<td>Disables single stepping at the current level and enables command stepping at the previous level (acts like <code>Break &quot;C&quot;</code> at the current level and <code>Break &quot;S&quot;</code> at the previous level).</td>
</tr>
<tr>
<td>Break &quot;L&quot;</td>
<td>Use <code>Break &quot;L&quot;</code> (Line Break) to step through your code a single routine line at a time, breaking at the beginning of every line. The system stops breaking when a <strong>Do</strong> command, an <strong>Xecute</strong> command, or user-defined function is encountered, and resumes when the command or function is done.</td>
</tr>
<tr>
<td>Break &quot;L+&quot;</td>
<td>Acts like <code>Break &quot;L&quot;</code>, except that Caché continues to break at the beginning of every routine line when a <strong>Do</strong> command, <strong>Xecute</strong> command, or user-defined function is encountered.</td>
</tr>
<tr>
<td>Break &quot;L-&quot;</td>
<td>Disables single stepping at the current level and enables line stepping at the previous level (acts like <code>Break &quot;C&quot;</code> at the current level and <code>Break &quot;L&quot;</code> at the previous level).</td>
</tr>
<tr>
<td>Break &quot;C&quot;</td>
<td>Use <code>Break &quot;C&quot;</code> (Clear Break) to stop breaking. Breaking will resume at a higher routine level after the job executes a <strong>Quit</strong> if a Break state is in effect at that higher level.</td>
</tr>
<tr>
<td>BREAK &quot;OFF&quot;</td>
<td>Use <code>Break &quot;OFF&quot;</code> to remove all debugging that has been established for the process.</td>
</tr>
</tbody>
</table>

Caché stacks the `Break` state whenever a **DO**, **XECUTE**, **FOR**, or user-defined function is entered. If you choose `B "C"` to turn off breaking, the system restores the `Break` state at the end of the **DO**, **XECUTE**, **FOR**, or user-defined function. Otherwise, Caché ignores the stacked state.

Thus if you enable breaking at a low subroutine level, breaking continues after the routine returns to a higher subroutine level. In contrast, if you disable breaking at a low subroutine
level that was in effect at a higher level, breaking resumes when you return to that higher level.

When you enter Programmer Mode, the **Break** state is not stacked. Thus you can change the **Break** state and the new state remains in effect when you issue an argumentless GOTO to return to the executing routine.

Periodic breaking does not occur for lines of code executed in Programmer Mode or for XECUTE lines started from Programmer Mode. When you enter Programmer Mode after a Break, you can enter Caché ObjectScript commands and use the line editor without encountering periodic breaking. However, after you issue a DO command or user-defined function, breaking resumes if B "L+" or B "S+" is in effect.

When B "L" or B "S" is in effect in Programmer Mode, a DO from Programmer Mode breaks on the first line or command in the routine, although a subsequent DO does not. Therefore, you can begin debugging by entering a B "L" or B "S" in Programmer Mode then issuing a DO without specifying B "L+" or B "S+".

### 16.2.4.1 Shutting Off Debugging

To remove all debugging that has been established for a process, use the **BREAK "OFF"** command. This command removes all breakpoints and watchpoints and turns off stepping at all program stack levels. It also removes the association with the debug and trace devices, but does not close them.

Invoking **BREAK "OFF"** is equivalent to issuing the following set of commands:

```
ZBREAK /CLEAR
ZBREAK /TRACE:OFF
ZBREAK /DEBUG:""
ZBREAK /ERRORTRAP:ON
BREAK "C" ; applied to all stack levels
```

### 16.2.5 Enabling Single Stepping at the Previous Execution Level

Use the **Break** command with “L-” or “S-” to end single stepping at the current execution and enable it at the previous execution level.

These are very similar to the “C” **Break** argument except that the "C" argument does not enable stepping at the previous level where single stepping may not have been activated yet.
16.2.6 Understanding the Programmer Mode Prompt Information

When a **Break** command suspends execution of a routine or when an error occurs, the program stack retains some stacked information. When this occurs in Programmer Mode, a brief summary of this information is displayed before the Programmer Mode prompt ( > ). Such messages take the form: 5d3>, where:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Indicate there are five <strong>DO</strong>, <strong>FOR</strong>, user-defined functions, <strong>XECUTE</strong>, <strong>ERROR</strong>, and <strong>BREAK</strong> states stored on the program stack.</td>
</tr>
<tr>
<td>d</td>
<td>Indicates that the last item stacked is a <strong>DO</strong>.</td>
</tr>
<tr>
<td>3</td>
<td>Indicates how many <strong>QUITs</strong> need to be performed in order to unstack the most recent <strong>NEW</strong> command, parameter passing, or user-defined function. This value is a zero if no <strong>NEW</strong> commands, parameter passing, or user-defined functions are stacked.</td>
</tr>
</tbody>
</table>

Prompts you can see in such messages are listed in the following table.

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td><strong>DO</strong></td>
</tr>
<tr>
<td>e</td>
<td><strong>user-defined Function</strong></td>
</tr>
<tr>
<td>f</td>
<td><strong>FOR Loop</strong></td>
</tr>
<tr>
<td>x</td>
<td><strong>XECUTE</strong></td>
</tr>
<tr>
<td>B</td>
<td><strong>Break</strong> state</td>
</tr>
<tr>
<td>E</td>
<td><strong>Error state</strong></td>
</tr>
<tr>
<td>S</td>
<td><strong>Sign on state</strong></td>
</tr>
</tbody>
</table>

16.2.7 Resuming Execution after a Break or an Error

When entering Programmer Mode after a **Break** or an error, Caché keeps track of the location of the command that caused the **Break** or error. Later, you can resume execution at the next command simply by entering an argumentless GOTO in Programmer Mode:

4f0>G
By typing a GOTO with an argument, you can resume execution at the beginning of another line in the same routine with the break or error, as follows:

```
4f0>G TAG
```

You can also resume execution at the beginning of a line in a different routine:

```
4f0>G TAB^ROU
```

Alternatively, you may clear the program stack with an argumentless QUIT command:

```
4f0>Q
```

**16.2.7.1 Sample Dialogs**

The following routines are used in the examples below.

```objectscript
MAIN ; 03 Jan 99 11:40 AM
  Set X=1,Y=6,Z=8
  Do ^SUB1 Write !,"SUM=",SUM
  Quit

SUB1 ; 03 Jan 99 11:42 AM
  Set SUM=X+Y+Z
  Quit
```

**With Break "L"**, breaking does not occur in the routine SUB1.

```
%SYS>Break "L"
%SYS>Do ^MAIN
  Set X=1,Y=6,Z=8
  ^
  <BREAK>MAIN+1^MAIN
  2d0>G
  Do ^SUB1 Write !,"SUM=",SUM
  ^
  <BREAK>MAIN+2^MAIN
  2d0>G
  SUM=15
  Quit
  ^
  <BREAK>MAIN+3^MAIN
  2d0>G
%SYS>
```

**With Break "L+"**, breaking also occurs in the routine SUB1.
16.2.8 The NEW Command in Programmer Mode

The argumentless NEW command effectively saves all symbols in the symbol table so you can proceed with an empty symbol table. You may find this command particularly valuable when you are in Programmer Mode after an error or Break.

To run other routines without disturbing the symbol table, issue an argumentless NEW command in Programmer Mode. The system then:

- Stacks the Programmer Mode frame on the program stack
- Reenters Programmer Mode.

For example:

```
4d0>N
5B1>D ^%T
3:49 PM
5B1>Q 1
4d0>G
```

The 5B1> prompt indicates that the system has stacked the Programmer Mode entered through a Break. The 1 indicates that a NEW command has stacked variable information, which you can remove by issuing a QUIT 1. When you wish to resume execution, issue a QUIT 1 to restore the old symbol table, and a GOTO to resume execution.

Whenever you use a NEW command, parameter passing, or user-defined function, the system places information on the stack indicating that later an explicit or implicit QUIT at the current subroutine or XECUTE level should delete certain variables and restore the value of others.
In Programmer Mode, you may find it useful to know if any NEW commands, parameter passing, or user-defined functions have been executed (thus stacking some variables), and if so, how far back on the stack this information resides.

### 16.2.9 The QUIT Command in Programmer Mode

In Programmer Mode, you can remove all items from the program stack by entering an argumentless QUIT command:

```
4f0>Q
%SYS>
```

To remove only a couple of items from the program stack (for example, to leave a currently executing subroutine and return to a previous DO level), use a QUIT with arguments. QUIT 1 removes the last item on the program stack, QUIT 3 removes the last three items, and so forth, as illustrated below:

```
9f0>Q 3
6d0>
```

### 16.2.10 Caché Error Messages

Caché displays error messages within angle brackets, as in `<ERROR>`, followed by a reference to the line that was executing at the time of the error and by the routine. A caret (^) separates the line reference and routine. Also displayed is the intermediate code line with a caret character under the first character of the command executing when the error occurred. For example:

```
S X=Y+3 D ^ABC
^<UNDEFINED>TAG+3^ROUT
```

This error message indicates an `<UNDEFINED>` error (that refers to the variable Y) in line TAG+3 of routine ROU. At this point, this message is also the value of the special variable $ZE.

### 16.3 Using %STACK to Display the Stack

You can use the %STACK utility to:

- Display the contents of the process execution stack
- Display the values of local variables, including values that have been "hidden" with the NEW command or through parameter passing.

- Display the values of process state variables, such as $IO and $JOB

### 16.3.1 Running %STACK

You execute %STACK by entering the following command:

```
%SYS>DO ^%STACK
```

As shown in this example, the %STACK utility displays the current process stack without variables.

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGN ON</td>
<td></td>
<td>~D ^StackTest</td>
</tr>
<tr>
<td>2</td>
<td>DO</td>
<td>TEST+1^StackTest</td>
<td>S A=1 ~D TEST1 Q ;level=2</td>
</tr>
<tr>
<td>3</td>
<td>NEW ALL/EXCL</td>
<td>NEW (E)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DO</td>
<td>TEST1+1^StackTest</td>
<td>~DO TEST2 ;level = 3</td>
</tr>
<tr>
<td>5</td>
<td>NEW</td>
<td></td>
<td>NEW A</td>
</tr>
<tr>
<td>6</td>
<td>DO</td>
<td>TEST1+1^StackTest</td>
<td>~DO TEST2 ;level = 3</td>
</tr>
<tr>
<td>7</td>
<td>ERROR TRAP</td>
<td>TEST2+2^StackTest</td>
<td>S $ZTRAP=&quot;TrapTag^StackTest&quot;</td>
</tr>
<tr>
<td>8</td>
<td>XECUTE</td>
<td>^StackTest</td>
<td>~X &quot;$A=$$TEST3()&quot;</td>
</tr>
<tr>
<td>9</td>
<td>$$EXTFUNC</td>
<td>^StackTest</td>
<td>~S A=$$TEST3()</td>
</tr>
<tr>
<td>10</td>
<td>PARAMETER</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DIRECT</td>
<td>TEST3+1^StackTest</td>
<td>~B</td>
</tr>
<tr>
<td>12</td>
<td>DO</td>
<td>^StackTest</td>
<td>~D ^%STACK</td>
</tr>
</tbody>
</table>

Under the current execution stack display, %STACK prompts you for a stack display action. You can get help by entering a question mark (?) at the “Stack Display Action” prompt.

### 16.3.2 Displaying the Process Execution Stack

Depending on what you enter at the “Stack Display Action” prompt, you can display the current process execution stack in four forms:

- Without variables, by entering *F
- With a specific variable, by entering *V
- With all variables, by entering *P
- With all variables, preceded by a list of process state variables, by entering *A

#### 16.3.2.1 Displaying the Stack without Variables

The process execution stack without variables appears when you first enter the %STACK utility or when you type the stack action *F.
16.3.2.2 Displaying the Stack with a Specific Variable

Enter *V at the "Stack Display Action" prompt, followed by the name of the variable you want to track through the stack. In the following example, the variable E is being tracked and the display is sent to the screen by pressing RETURN at the “Device:” prompt.

Stack Display Action: *V
Variable(s): E
Display on Device: <RETURN>

16.3.2.3 Displaying the Stack with All Defined Variables

Enter *P to see the process execution stack together with the current values of all defined variables.

16.3.2.4 Displaying the Stack with All Variables, including State Variables

You can print all possible reports to screen, file or printer by entering *A at the "Stack Display Action" prompt. This report prints the following:

- Process state variables
- Process execution stack with all variables

16.3.3 Understanding the Stack Display

Each item on the stack is called a frame. The following table describes the information provided for each frame.
%STACK Utility Information

### Description

<table>
<thead>
<tr>
<th>Heading</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Identifies the level within the stack. The oldest item on the stack is number 1. Frames without an associated level number share the level that first appears above them.</td>
</tr>
<tr>
<td>Type</td>
<td>Identifies the type of frame on the stack, which can be: DIRECT Break A Break command was encountered that caused a return to direct mode. DIRECT CALLIN A Caché process was initiated from an application outside of Caché, using the Caché call-in interface. DIRECT ERROR An error was encountered that caused a return to direct mode. DO A DO command was executed. ERROR TRAP If a routine sets $ZTRAP, this frame identifies the location where an error will cause execution to continue. FOR A FOR command was executed. NEW A NEW command was executed. If the NEW command had arguments, they are shown. SIGN ON Execution of the Caché process was initiated. XECUTE An XECUTE command was executed. $$EXTFUNC A user-defined function was executed.</td>
</tr>
<tr>
<td>Line</td>
<td>Identifies the Caché ObjectScript source line associated with the frame, if available, in the format tag+offset^routine.</td>
</tr>
<tr>
<td>Source</td>
<td>Shows the source code for the line, if it is available. If the source is too long to display in the area provided, horizontal scrolling is available. If the device is line-oriented, the source wraps around and continued lines are preceded with “...”.</td>
</tr>
</tbody>
</table>

The following table shows whether level, line, and source values are available for each frame type. A "No" under Level indicates that the level number is not incremented and no level number appears in the display.

### Frame Types and Values Available

<table>
<thead>
<tr>
<th>Frame Type</th>
<th>Level</th>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT Break</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DIRECT CALL IN</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DIRECT ERROR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DO</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
</tr>
<tr>
<td>ERROR TRAP</td>
<td>No</td>
<td>No</td>
<td>No, but the new $ZT value is shown.</td>
</tr>
<tr>
<td>FOR</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### 16.3.3.1 Moving through %STACK Display

If a %STACK display fills more than one screen, you see the prompt “-- more --” in the bottom left corner of the screen. At the last page, you see the prompt “-- fini --” . Type ? to see key presses you use to maneuver through the %STACK display.

- - - Filter Help - - -
<space> Display next page.
<return> Display one more line.
T Return to the beginning of the output.
B Back up one page (or many if arg>1).
R Redraw the current page.
/text Search for \\qtext\\q after the current page.
A View all the remaining text.
Q Quit.
? Display this screen
# specify an argument for B, L, or W actions.
L set the page length to the current argument.
W set the page width to the current argument.

You enter any of the commands listed above whenever you see the “-- more --” or “-- fini --” prompts.

For the B, L and W commands, you enter a numeric argument prior to the command letter. For instance, enter 2B to move back two pages, or enter 20L to set the page length to 20 lines.

---

<table>
<thead>
<tr>
<th>Frame Type</th>
<th>Level</th>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>No</td>
<td>No</td>
<td>Shows the form of the NEW (inclusive or exclusive) and the variables affected.</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>No</td>
<td>No</td>
<td>Shows the formal parameter list. If a parameter is passed by reference, shows what other variables point to the same memory location.</td>
</tr>
<tr>
<td>SIGN ON</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>XEXECUTE</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
</tr>
<tr>
<td>$$EXTFUNC</td>
<td>Yes</td>
<td>Yes*</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* The LINE value is blank if these are invoked from Programmer Mode.
16.3.3.2 Displaying Variables at Specific Stack Level

To see the variables that exist at a given stack frame level, enter `?#` at the “Stack Display Action” prompt, where `#` is the stack frame level. The following example shows the display if you request the variables at level 1.

```
Stack Display Action: ?1
The following Variables are defined for Stack Level: 1
E
```

16.3.3.3 Displaying Stack Levels with Variables

You can display the variables defined at all stack levels by entering `??` at the “Stack Display Action” prompt. The following example shows a sample display if you select this action.

```
Stack Display Action: ??
Now loading variable information ... 19
Base Stack Level: 5
A
Base Stack Level: 3
A B C D
Base Stack Level: 1
E
```

16.3.3.4 Displaying Process State Variables

To display the process state variables, such as $IO, enter `*S` at the "Stack Display Action" prompt. You will see the defined variables as shown in the following figure. This display may appear on two screens on your terminal.
Process State Intrinsics:

$D = $EL = $ES = $H = 55574,43548
$I = TTA0:
$J = 20592
$K = $P = $R = $S = 236016
$SY = $T = 0
$TL = $TR = $X = 0
$Y = 14
$ZA = 0
$ZB = $c(13)
$ZI = <UNDEFINED>RestST+3^%STACK
$ZR = ^mtemp(33)
$ZT = TESTQ^%STACKD
$ZU(100) =
$ZU(12) = DUA0: ["sys1"]
$ZU(18) = 0
$ZU(20) = DUA0: ["sys1"]
$ZU(39) = DUA0: ["sys1"]
$ZU(5) = DUA0: ["sys1"]
$ZU(55) = 0
$ZU(68,1) = 0
$ZU(68,5) = 1
$ZU(68,6) = 0
$ZU(68,7) = 0
--fini--

16.3.3.5 Printing the Stack and/or Variables

When you select the following actions, you can choose the output device:

- *P
- *A
- *V after selecting the variables you want to display.

16.4 Error Trap Utilities

The error trap utilities, %ETN and %ERN, help in error analysis by storing variables and recording other pertinent information about an error.
16.4.1 %ETN Application Error Trap

When you set the reserved variable $ZT to an entry reference (such as TAG^ROU), errors adjust the stack and GOTO to that location. You may find it convenient to set the error trap to execute the utility %ETN on an application error. This utility saves valuable information about the job at the time of the error. You can later call the %ERN error report utility to examine the information. Use the following code to set the error trap to this utility:

```
SET $ZT="^%ETN"
```

When an error occurs and you call the %ETN utility, you see a message similar to the following message:

```
Error has occurred: <SYNTAX> at 10:30 AM
```

You may find it useful to set an error trap in an application routine only if it is used in Application Mode (rather than in Programmer Mode). The following code sets an error trap only if Caché is in Application Mode:

```
SET $ZT=$S($ZJ#2:"",1:"^%ETN")
```

16.4.2 %ERN Application Error Report

The %ERN utility examines application errors recorded by the %ETN error trap utility.

Take the following steps to use the %ERN utility:

1. When prompted, enter the date on which the errors occurred or enter a question mark (?). If you enter “?” in response to the “Enter date:” prompt, you get a list of dates and the number of errors on each date. When entering a date, use any date format that is accepted by the %DATE utility.

2. When prompted for the error you wish to examine, supply the number of the error you want (1 for the first error, 2 for the second, and so on) or enter a question mark (?) for a list of available responses.

   The utility displays information about the error, including the line of code executed at the time of the error.

3. You are then prompted for a variable. You can now enter either a question mark (for a list of further options) or one of the responses shown in the following table:
### %ERN Options

<table>
<thead>
<tr>
<th>%ERN Option</th>
<th>Action or Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>A complete list of options is displayed.</td>
</tr>
<tr>
<td>*C</td>
<td>You may enter a comment in the error log.</td>
</tr>
<tr>
<td>*F</td>
<td>The execution frame stack is displayed</td>
</tr>
<tr>
<td>*L</td>
<td>All the variables are loaded into your partition.</td>
</tr>
<tr>
<td>*S</td>
<td>All the process state variables are displayed.</td>
</tr>
<tr>
<td>variable name</td>
<td>If you enter the name of a variable, the value of the variable will be displayed. You can examine only nonsubscripted variables this way.</td>
</tr>
</tbody>
</table>

4. If you press **RETURN** in response to the Variable: prompt, you return to the Error #: prompt.

5. If you press **RETURN** again, you return to the For Date: prompt.

6. Press **RETURN** a third time to leave the utility.

7. If you are prompted to delete old errors, answer **Y** or press **RETURN**.

In the following code, a ZLOAD of the routine REPORT is issued to illustrate that by loading all of the variables with "*LOAD" and then loading the routine, you can recreate the state of the job when the error occurred except that the program stack, which records information about DOs, etc., is empty.
Command-Line Routine Debugging

%SYS>D ^%ERN

For date:4/30/99  3 errors.

Error: ?L

1) <DIVIDE>CALC+4^CALC at 01:35PM. Device=70, TRM #70.
   $ZA=0, $ZB="^M", $ZS=20
   S C=R/(F+DT)

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16PM. Device=70, TRM #70.
   ZA=0, $ZB="^M", $ZS=20
   S ^REPORT(%DAT,TYPE)=I

3) <SYNTAX>ZSET+5^ZSET at 10:34AM. Device=70, TRM #70.
   $ZA=0, $ZB="^M", $ZS=20
   X XSET

Error: 2

2) <SUBSCRIPT>REPORT+4^REPORT at 03:16PM. Device=70, TRM #70.
   ZA=0, $ZB="^M", $ZS=20
   S ^REPORT(%DAT,TYPE)=I

Variable: %DAT
   %DAT="Apr 30 99"

Variable: TYPE
   TYPE=""

Variable: *LOAD
%SYS>ZL REPORT

%SYS>W

%DAT="Apr 30 99"
%DS=""
%TG="REPORT+1"
I=88
TYPE=""
XY="S $X=250 W *27,*91,DY+1,*59,DX+1,*72 S $X=DX,$Y=DY"
%SYS>
This chapter details usage of a dll to check the status of Caché and perform calls to Caché from a Windows client.

Caché provides a mechanism for Windows client programs to control a Caché configuration and to start up Caché processes. This allows you to deliver applications that automatically start Caché processes with the correct configuration information without requiring the standard Caché tools. The tools allow you to:

- Find Caché directories paths and service name.
- Get the status of the Caché system.
- Control a Caché configuration directly or through the Caché Control Service, depending on which version of Windows is running
- Start a Caché process with the appropriate settings.

Caché provides sample programs in C, C++, and Visual Basic that demonstrate dynamic loading of cctrldll and use of the functions to start, stop, and force a configuration, and to start Caché processes. These samples are located in the Dev/cache subdirectory of your Caché installation.

- cctrlecpp—C++ code sample
- cctrlvb—Visual Basic code sample
- ctrldemo—C code sample
17.1 CctrlGetDirs

Finds configuration, binary, and manager directory paths, and service name for a given configuration name.

17.1.1 Syntax

CctrlGetDirs(char *config, CCTRL_DIR_INFO *dirinfo)

<table>
<thead>
<tr>
<th>config</th>
<th>The name of the desired configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirinfo</td>
<td>A pointer to a C structure where the directory information will be stored.</td>
</tr>
</tbody>
</table>

17.1.2 Return Value

Returns (char *0) on ERROR.

17.2 CctrlConfigStatus

Returns the status of the Caché configuration.

17.2.1 Syntax

CctrlConfigStatus(char* config)

| config | The name of the desired configuration |

17.2.2 Return Values

Returns a value from 0 through 4 as follows:
### 17.3 CctrlControl

Controls a Caché configuration through the Caché Control Service on Windows NT, or directly on Windows 95/98.

#### 17.3.1 Syntax

`CctrlControl(char *command, char *config)`

<table>
<thead>
<tr>
<th><code>command</code></th>
<th>Use one of the following commands:</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>starts a configuration</td>
</tr>
<tr>
<td>stop</td>
<td>shuts down a configuration</td>
</tr>
<tr>
<td>stopnoshut</td>
<td>shuts down a configuration without running the user-supplied shutdown routine</td>
</tr>
<tr>
<td>force</td>
<td>forces down a configuration; equivalent to <code>cforce</code> on UNIX systems</td>
</tr>
<tr>
<td>stopstart</td>
<td>shuts down a configuration gracefully and immediately restarts it</td>
</tr>
</tbody>
</table>

| `config`  | The name of the desired configuration. |

<table>
<thead>
<tr>
<th></th>
<th>Configuration is up and running.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Configuration is starting or stopping.</td>
</tr>
<tr>
<td>2</td>
<td>Configuration startup or shutdown aborted.</td>
</tr>
<tr>
<td>3</td>
<td>Configuration is down.</td>
</tr>
<tr>
<td>4</td>
<td>ERROR</td>
</tr>
</tbody>
</table>
17.3.2 Return Values

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCTRL_SUCCESS</td>
<td>Operation succeeded</td>
</tr>
<tr>
<td>CCTRL_ERROR</td>
<td>Generic error</td>
</tr>
<tr>
<td>CCTRL_INVALID_COMMAND</td>
<td>Invalid command argument</td>
</tr>
<tr>
<td>CCTRL_INVALID_CONFIGURATION</td>
<td>Undefined configuration</td>
</tr>
<tr>
<td>CCTRL_CONTROL_STU_ERROR</td>
<td>^STU failed</td>
</tr>
</tbody>
</table>

Following an error return, CctrlGetLastError returns a pointer to an informational error string.

17.4 CctrlRun

Starts a Caché process in the indicated configuration, and namespace, using the indicated principal I/O device and invoking the indicated routine.

17.4.1 Syntax

CctrlRun(char *config, char *routine, char *namespace, char *IOType)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>The name of the running configuration.</td>
</tr>
<tr>
<td>routine</td>
<td>The name of the desired routine to start.</td>
</tr>
<tr>
<td>namespace</td>
<td>The name of the desired namespace.</td>
</tr>
<tr>
<td>IOType</td>
<td>How I/O is to be handled, which can have a value of either cterminal or none.</td>
</tr>
<tr>
<td></td>
<td>• cterminal—Caché programmer's terminal.</td>
</tr>
<tr>
<td></td>
<td>• none—No I/O. Process will run in the background with NUL: used for $Principal. Writes to $Principal are discarded. Reads from $Principal produce an error.</td>
</tr>
</tbody>
</table>
17.4.2 Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCTROL_SUCCESS</td>
<td>Operation succeeded</td>
</tr>
<tr>
<td>CCTROL_ERROR</td>
<td>Generic error</td>
</tr>
<tr>
<td>CCTROL_INVALID_COMMAND</td>
<td>Invalid command argument</td>
</tr>
<tr>
<td>CCTROL_INVALID_CONFIGURATION</td>
<td>Undefined configuration</td>
</tr>
<tr>
<td>CCTROL_CONTROL_STU_ERROR</td>
<td>^STU failed</td>
</tr>
</tbody>
</table>

**Note:** On Windows NT, the specified configuration must be running. If you are not sure if the configuration is running, use CctrlConfigStatus and CctrlControl to check and start the desired configuration. This prevents Caché from trying to start a configuration without using the control service.

17.5 CctrlRunIO

Starts a Caché process in the indicated configuration, and namespace, using the indicated principal I/O device type, invoking the indicated routine and additional IO specifications for input, output and error devices.

17.5.1 Syntax

CctrlRunIO(
    char *config,
    char *routine,
    char *namespace,
    char *IOType,
    HANDLE *hIO,
    char *cwd,
    char *options,
    HANDLE *child,
    DWORD *childPID))
| **config** | The name of the running configuration in all capital letters. |
| **routine** | The name of the desired routine to start. |
| **namespace** | The name of the desired namespace. |
| **IOtype** | How I/O is to be handled, which can have a value of TCP, cterminal, or none:  
  - TCP—TCP socket.  
  - cterminal—Caché programmer's terminal.  
  - none—No I/O. Process will run in the background with NUL: used for $Principal. Writes to $Principal are discarded. Reads from $Principal produce an error. |
| **hIO** | An array of three handles to be used as the standard input, output, and error devices of the Caché process. |
| **cwd** | The working directory path of the child process. If the directory argument is zero, the working directory of the current process is used. |
| **option** | Additional cache.exe command line options appended to the generated command line. For example, you could define a larger process memory size (-b 1024). |
| **child** | The pointer to a variable of type HANDLE, where the handle to the child process will be returned. If the value of handle is zero, the handle to the child process will be closed by this function. |
| **childPID** | The pointer to the PID of the created cache.exe process. This argument can be zero if the child's PID is not required. |

### 17.5.2 Return Values

| **CCTRL_SUCCESS** | Operation succeeded |
| **CCTRL_ERROR** | Generic error |
| **CCTRL_INVALID_COMMAND** | Invalid command argument |
| **CCTRL_INVALID_CONFIGURATION** | Undefined configuration |
| **CCTRL_CONTROL_STU_ERROR** | ^STU failed |
Note: The handles in the hIO array must be inheritable. Use DuplicateHandle to make the handle inheritable by the child process.

On Windows NT, the specified configuration must be running. If you are not sure if the configuration is running, use CctrlConfigStatus and CctrlControl to check and start the desired configuration. This prevents Caché from trying to start a configuration without using the control service.
Caché offers language compatibility modes for:

- DSM-11
- Open M [DSM]
- Open M [DTM]
- Open M [MSM]

These modes make it possible to port applications from the above installations into Caché by accommodating the syntax of many of the more common Open M commands and functions.

This chapter presents the following topics as they relate to compatibility modes:

- Displaying and Switching Language Mode
- DSM-11 Language Compatibility
- DSM Language Compatibility
- DSM-J Language Compatibility
- DTM Language Compatibility
- MSM Language Compatibility
18.1 Displaying and Switching Language Mode

Caché provides a $ZU function that allows you to display and set the language mode. $ZU(55, n) sets the language to mode n and returns the previous value. $ZU(55) returns the current value.

Do not modify the current dialect using $ZU(55) when there is already a routine in the partition created under a different dialect.

**Note:** You cannot set the language mode from the GUI utilities. You must use the $ZU(55) function.

If you follow the procedures outlined in this section to port, manage and convert DSM mode routines, you do not need to use $ZU(55,n) calls to set language mode. If you want to use $ZU(55) to see the current language mode, issue the following command:

```
Set X=$ZU(55)
Write X
```

You can also set the language mode by any command that affects loading of a routine. The current dialect will be set to that of the routine. In the case of a Do, the previous dialect is restored upon leaving the routine. This will also happen for ZLoad or GoTo done within command indirection (Xecute).

The general-use language modes represented in Caché are as follows.

### Caché Language Modes

<table>
<thead>
<tr>
<th>$ZU(55) Value</th>
<th>Language Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caché</td>
</tr>
<tr>
<td>1</td>
<td>DSM-11</td>
</tr>
<tr>
<td>2</td>
<td>DTM</td>
</tr>
<tr>
<td>5</td>
<td>DSM</td>
</tr>
<tr>
<td>6</td>
<td>DSM-J (Japanese version)</td>
</tr>
<tr>
<td>7</td>
<td>DTM-J (Japanese version)</td>
</tr>
<tr>
<td>8</td>
<td>MSM</td>
</tr>
</tbody>
</table>
18.2 DSM-11 Language Compatibility

Caché operates in DSM mode when working with a DSM-11 routine that has been ported. You need not make extensive changes in order to run DSM-11 routines that have been ported to Caché. This section discusses how various commands and functions operate in DSM-11 mode.

DSM-11 Language mode is $ZU(55,1).

Note: Pay particular attention to Lock, ZAllocate, and ZDeallocate.

18.2.1 Using Routine Interlock Devices

Routine interlock devices provide an alternative to the standard M Lock command. Routine interlock devices were developed so DSM-11 applications would not run out of space in the DSM-11 Lock Table.

This functionality is especially designed for DSM-11 compatibility mode, with applications ported from DSM-11. However, it operates in Caché language mode as well.

18.2.1.1 Opening Special Devices to Lock a Global

Device numbers 20 through 46 and 200 through 223 serve as routine interlock devices. They are pseudodevices that processes use cooperatively, so that they can regulate updates to the same data. The routine interlock device numbers exist on all Open M and Caché systems.

Two or more processes must agree to use one of these device numbers for a particular purpose. One process issues an Open command for that device, and performs any necessary processing on data associated with that device. If a second process tries to open that device, it must wait until the first one closes the device. Consequently, the second process performs no processing until it successfully opens the device or times out. Any number of processes can attempt to open the device; each one waits until the one before it releases the device with a Close command.

In this way, an Open command to a routine interlock device guarantees a process exclusive access to the data associated with the device, as long as all applications:

- Are consistent in their use of this technique (as with the Lock command)
- Agree on the data associated with each routine interlock device
After the process issues a Close command to that device number, another process can open the device and have exclusive access to the data.

### 18.2.2 Issuing I/O Commands for Routine Interlock Devices

**Open** and **Close** are the only I/O commands you can issue to routine interlock devices. There are no **Read** and **Write** commands, because using a routine interlock device does not involve transferring data in or out of any device or memory area.

#### 18.2.2.1 Open Command

The **Open** command opens a routine interlock device and prevents other processes from successfully issuing an **Open** command to the same device number.

**Syntax**

```
Open device[:timeout]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>A device number, from 20 through 46 or 200 through 223. You must ensure that all routines that use a given device number agree on the meaning of that number. The meaning is usually which global(s) are now considered locked and thus not available to other processes.</td>
</tr>
<tr>
<td>timeout</td>
<td>A positive integer whose value in seconds is the longest time Caché waits for an Open to finish. If you specify 0, the Open returns control to the process immediately.</td>
</tr>
</tbody>
</table>

#### 18.2.2.2 Close Command

The **Close** command closes a routine interlock device and releases it to other processes.

**Syntax**

```
Close device
```

### 18.2.3 Working with DSM-11 Mode Routines

Keep the following points in mind when working with DSM-11 mode routines on a Caché system:

- DSM-11 and InterSystems routines can coexist in the same namespace.
• Caché and DSM-11 mode routines can call one another as desired. Caché adapts to the saved language mode of the executing routine on a per routine basis.

• DSM11 mode routines can only be saved as intermediate source code (.INT) routines. They cannot be manipulated at the macro source code (.MAC) level.

• Caché provides only runtime support for the DSM-11 language. It does not support the DSM-11 development environment, nor does it support calls to the DSM-11 library utilities or MACRO-11 routines.

• You can only do further development on DSM-11 routines in the Caché development environment.

• When you load a routine with the Full Screen Editor or the ZLoad command, Caché sets the language mode to match that of the routine.

• If you change the language mode with the $ZU(55,n) switch so that it does not match that of the loaded routine, and then attempt to do a ZInsert, you will get a <LANGUAGE MISMATCH> error message.

### 18.2.3.1 Converting a Routine from DSM-11 Mode to Caché Mode

At any time, you can edit a DSM-11 routine to conform to Caché and change its language mode to Caché. The Caché Studio editor facilitates the conversion of DSM-11 mode routines to Caché mode.

### 18.2.3.2 Routine Manipulation

#### Saving and Restoring Routines

After you have DSM-11 mode routines on your Caché system, you can use the utilities %RO and %RI or Caché Studio with them. These utilities maintain the DSM-11 mode of the routines, which lets allows you copy routines that you have copied from a DSM11 host to a Caché host to additional Caché namespaces and/or hosts using the Caché %RO and %RI utilities.

#### Listing Routines in the Current Namespace

The %RD utility, which lists the routines in the current Caché namespace, includes a column “LANG” in its long form display that reflects the saved language mode of each routine. The “LANG” column is empty for Caché mode routines and contains “DSM11” for DSM11 mode routines. You can also use the Caché Explorer utility to list routines.
The %RCOPY and %RCOMPIL routine utilities recognize and preserve the language mode of each routine. %RCOPY copies routines and generates backup versions. %RCOMPIL compiles macro source and intermediate code routines.

Use %RCOPY to rename a DSM11 mode routine without converting it to Caché mode. You can also use the Caché Studio.

**Caché Studio and Routine Line Editor**

The Caché Studio and the Routine Line Editor (X^%) recognize and preserve the language mode of each routine, if you SAVE and COMPILE. The Routine Properties dialogue in the Caché Studio displays the language mode of the current routine.

**18.2.4 Transferring Globals from DSM-11 Systems**

**18.2.4.1 ANSI-Collated Globals**

To transfer ANSI-collated globals from a DSM-11 system to an Caché system, first read them from the DSM-11 system using the Caché Studio with the DSM-11 file format.

**18.2.4.2 String-Collated Globals**

To transfer string-collated globals from a DSM11 system to Caché, first read the globals from the DSM-11 system using the DSM11 %GTO utility. Then, use the %GCREATE utility on the InterSystems host to create each global and give it the pure string collation characteristic. Finally, load the globals in using the %GIGEN utility.

**18.2.5 I/O Programming in DSM-11 Compatibility Mode**

The following sections discuss input and output programming characteristics of Caché’s DSM-11 compatibility mode.

**18.2.5.1 Terminal I/O in DSM-11 Compatibility Mode**

In Caché, the Open and Use commands for terminals offer only three parameters: margin, protocol, and terminators.

By contrast, in DSM-11 the Open and Use commands accommodate eleven parameters. Most of these parameters exist because DSM-11 also serves as its own operating system. In DSM-11 compatibility mode, Caché simulates the DSM-11 syntax, accepting but ignoring parameters that relate to the operating system.
The following two tables describe the Caché DSM-11 compatibility mode interpretation of the eleven DSM-11 Open and Use command parameters.

### DSM-11 Compatibility Mode Open/Use Command Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>DSM-11 Meaning</th>
<th>DSM-11 Compatibility Mode Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: margin</td>
<td>Set right margin for device.</td>
<td>Same as DSM-11. Identical to Caché mode margin parameter both in functionality and parameter position.</td>
</tr>
<tr>
<td>p2: output ring buffer</td>
<td>Set the size of the output ring buffer.</td>
<td>Accepted but ignored.</td>
</tr>
<tr>
<td>p3: input field length</td>
<td>Specify field length. This parameter serves only for DSM-11 backward compatibility. The Read command is now the preferred method for specifying field length.</td>
<td>Accepted but ignored.</td>
</tr>
<tr>
<td>p4: input ring buffer</td>
<td>Set the size of the input ring buffer.</td>
<td>Accepted but ignored.</td>
</tr>
<tr>
<td>p5: set status</td>
<td>Modify device characteristics</td>
<td>Bit mask related to InterSystems mode protocol parameter. See next table for details.</td>
</tr>
<tr>
<td>p6: clear status</td>
<td>Modify device characteristics</td>
<td>Bit mask related to Caché mode protocol parameter.</td>
</tr>
<tr>
<td>p7: set $X and $Y</td>
<td>Change $X and $Y settings for the terminal.</td>
<td>See next table for details.</td>
</tr>
<tr>
<td>p8: line parameter register</td>
<td>Values assigned based on the type of controller.</td>
<td>Accepted but ignored.</td>
</tr>
</tbody>
</table>
The protocol parameter in Caché mode is a string containing some combination of the following:

- **B** (enable **Ctrl-C**)
- **C** (CRT)
- **F** (flush)
- **I** (image)
- **P** (printer)
- **S** (secret, no echo)
- **T** (terminators).

Similarly, DSM-11 has the set status and clear status parameters, both of which are bit masks.

To turn on a protocol, you must include its bit in the set status parameter. To turn off a protocol, you must include its bit in the clear status parameter. In DSM-11 compatibility mode, Caché accepts but ignores bits that are operating system specific. The following table lists the bits that are supported.

**DSM-11 Compatibility Mode Set and Clear Status Bits**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>0 - Echo; 1 - No Echo</td>
<td>Identical to Caché “S” protocol.</td>
</tr>
<tr>
<td>2 (4)</td>
<td>0 - Hardcopy; 1 - CRT</td>
<td>Identical to Caché “C” protocol.</td>
</tr>
<tr>
<td>Bit</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6 (64)</td>
<td>0 - Escape Processing Disabled; 1 - Escape Processing Enabled</td>
<td>Determines how escape sequences are handled on <strong>Read</strong>. In Caché mode, an escape sequence always terminates a non image mode <strong>Read</strong>, and the escape sequence appears as a string in <strong>$ZB</strong>. In DSM 11 mode, an escape sequence always terminates a normal <strong>Read</strong>, but the escape sequence is processed only if escape processing is enabled. When escape processing is enabled, the decimal ASCII code of the second character appears in the high byte of <strong>$ZB</strong> and the decimal ASCII code of the escape character itself appears in the low byte. When escape processing is disabled, the escape character is treated like any terminator and the remaining character(s) of the escape sequence are not processed. They remain in the input buffer where they can be accessed by subsequent <strong>Read</strong> commands.</td>
</tr>
<tr>
<td>7 (128)</td>
<td>0 - Do Not Transmit Cursor Control Sequences; 1 - Transmit Cursor Control Sequences</td>
<td>Determines the handling of parameter 7: set <strong>$X</strong> and <strong>$Y</strong>. When a parameter 7 value is supplied on <strong>Open/Use</strong> in DSM-11 language compatibility mode, <strong>$X</strong> and <strong>$Y</strong> are updated accordingly and if bit 7 is ON, the appropriate cursor control sequence (VT52 or ANSI, depending on bit 16) is transmitted to the terminal.</td>
</tr>
<tr>
<td>14 (16384)</td>
<td>0 - No conversion; 1 - Convert to uppercase</td>
<td>Determines whether or not case is converted on input.</td>
</tr>
<tr>
<td>16 (65536)</td>
<td>0 - VT52; 1 - ANSI</td>
<td>Determines the appropriate cursor control sequence.</td>
</tr>
<tr>
<td>19 (524288)</td>
<td>0 - Ignore Delete Character; 1 - Acknowledge Delete Character</td>
<td>Determines the effect of pressing the <strong>Delete</strong> key when there is nothing in the input buffer to delete. If the bit is ON, the delete character is ignored. If the bit is OFF, the <strong>Read</strong> terminates and the delete character appears as 127 in the low byte of <strong>$ZB</strong>.</td>
</tr>
<tr>
<td>Bit</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>20 (1048576)</td>
<td>0 - Accept non-programmed ctrl keys; 1 - Ignore non-programmed ctrl keys.</td>
<td>Determines the effect of pressing non-programmed control keys. When the bit is OFF, all non-programmed control keys are accepted as normal characters. When the bit is ON, non-programmed control keys are ignored. They are not echoed or placed in the input buffer.</td>
</tr>
<tr>
<td>23 (8388608)</td>
<td>0 - Disable Pass All; 1 - Enable Pass All</td>
<td>Identical to Caché IMAGE mode protocol. When the bit is ON, all control characters pass directly to the program without filtering or interpretation.</td>
</tr>
<tr>
<td>25 (33554432)</td>
<td>0 - Enable Type-ahead; 1 - Disable Type-ahead</td>
<td>Determines the handling of type-ahead. When bit 25 is ON, type-ahead is disabled. The input buffer is flushed before each <code>Read</code>. When bit 25 is OFF, type-ahead is enabled. The input buffer is not flushed before each <code>Read</code>.</td>
</tr>
<tr>
<td>26 (68108864)</td>
<td>0 - Default to Bit 25, Type-ahead control; 1 - Do not flush Type-ahead Buffer.</td>
<td>When bit 26 is OFF, no additional type-ahead control beyond that provided by bit 25 is conferred. When bit 26 is on, the type-ahead buffer can never be flushed, unless explicitly directed by bit 25. For example, <code>Read</code> with a prompt does not flush type-ahead.</td>
</tr>
</tbody>
</table>

On Windows, Caché ObjectScript allocates each process an open file quota between database files and files opened with **Open**. When **Open** causes too many files to be allocated to **Open** commands, you receive a `<TOOMANYFILES>` error message.

**CAUTION:** **Open/Use** parameters that are unique to DSM11 compatibility mode are honored at **Read/Write** time only if the last **Open/Use** of the device was in DSM-11 mode. The parameters unique to DSM-11 mode are p5/p6 (bits 6,7,19,20,25,26) and p7.

**Terminating a READ**

If a terminator completes a **Read**, the special variable **$ZB** contains the terminator character as a string in Caché mode, while in DSM-11 mode the ASCII decimal value of the terminator is returned in the low byte of **$ZB**.
If an escape sequence terminates a Read, Caché mode returns the ASCII escape sequence as a string in $ZB, while DSM-11 mode returns an integer with the escape character (27) in the low byte of $ZB and the second character plus 16 then modulo 64 in the high byte.

### 18.2.5.2 Magnetic Tape I/O

The I/O commands for magnetic tape are very similar in Caché and DSM-11. Both use the positional parameter style, and the positions and values of the specific parameters in each are quite similar. However, there are some differences in format codes, which account for these unique characteristics of magnetic tape I/O in DSM-11 compatibility mode:

- Caché does not support the DSM-11 “B” format.
- DSM-11 offers a “C” format, which specifies continuous (non-pending) I/O. This enables the program to go on and do other things while the tape I/O is in progress. You can check the status of the I/O at any time by issuing the `Write *10` command, which updates $ZA. While Caché cannot actually do this on the OpenVMS, UNIX, or Windows operating systems, it does accept the “C” format in DSM-11 compatibility mode so that DSM-11 applications encoded with it do not require source changes to run. Instead, Caché performs the following steps:
  - Performs the I/O.
  - Waits for the program to complete and then move on.
  - Updates $ZA with the completion status when the program issues the `Write *10`.
- The DSM-11 “T” format specifies tape mark trap inhibiting in DSM-11 compatibility mode.
- The format string characters used to specify tape density for DSM-11 and Caché relate as follows:

<table>
<thead>
<tr>
<th>DSM-11 mode</th>
<th>Caché mode</th>
<th>Tape Density (BPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>800</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1600</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6250</td>
</tr>
</tbody>
</table>

- In DSM-11 mode, the $ZA and $ZB special variables return magnetic tape operation values that are compatible with DSM-11.
- In Caché mode, magnetic tape control functions are accomplished by using `Write * -n` (negative value), while DSM11 compatibility mode uses `Write * n` (positive value).
The assignments of the absolute values to control functions are the same for both modes, except that DSM11 offers `Write *10`, which updates $ZA. This is useful with the DSM11 “C” format option for checking the status of the asynchronous I/O operation, as discussed above.

### 18.2.5.3 Interjob Communications Devices (JOBCOM)

Caché offers devices 224255 for interjob communications in both language modes. For more information on how to use these devices, see the chapter “Interprocess Communication.”

### 18.2.6 View Command and $View Function

Caché does not support the `View` command or the $View function in DSM11 compatibility mode.

### 18.2.7 ZAllocate and ZDeallocate

The `ZAllocate` and `ZDeallocate` commands behave differently in Caché and DSM-11:

- In the DSM-11 mode, `ZAllocate` can only be undone by `ZDeallocate` and `LOCK+` can only be undone by `Lock`. Moreover, any number of `ZAllocate` commands of a certain location can be undone with a single `ZDeallocate`.

- In Caché mode, `ZAllocate` and `ZDeallocate` are synonymous with `Lock+` and `Lock`.

Two switches in the `$ZU(68)` and `$ZU(69)` functions allow you to select which behavior `ZAllocate` and `ZDeallocate` enact. Switch 8 affects Caché mode routines and Switch 9 affects DSM-11 mode routines.

#### 18.2.7.1 DSM-11 Mode Routines

Set switch 9 to cause all `ZA` and `ZD` commands in DSM-11 compiled routines to act in Caché mode.

\[
\begin{align*}
$ZU(69,9,1) & \text{ System level} \\
$ZU(68,9,1) & \text{ Process level} \\
\end{align*}
\]

Clear switch 9 to cause all `ZA` and `ZD` commands in DSM-11 compiled routines to act in DSM-11 mode (this is the default).

\[
\begin{align*}
$ZU(69,9,0) & \text{ System level} \\
$ZU(68,9,0) & \text{ Process level} \\
\end{align*}
\]
18.2.7.2 Caché Mode Routines

Set switch 8 to cause all **ZA** and **ZD** commands in Caché mode routines to act in DSM-11 mode.

\[^{\$ZU(69,8,1)}\] System level
\[^{\$ZU(68,8,1)}\] Process level

Clear switch 8 to cause all **ZA** and **ZD** commands in Caché mode routines to act in Caché mode (this is the default).

\[^{\$ZU(69,8,0)}\] System level
\[^{\$ZU(68,8,0)}\] Process level

18.2.7.3 Determining Which Mode to Use

In DSM-11 compatibility mode, **ZAllocate** and **ZDeallocate** by default behave as they do in DSM-11 (unless you set switch 9). Because of the difference between these modes, always examine code you are porting from a DSM11 system to ensure that changes in nesting characteristics of **Lock** and **ZAllocate** will not affect the integrity of your application.

**Set Mode in ZSTU**

If you have ported your **ZA** and **ZD** commands to Caché mode, but are still compiling in DSM-11 mode, set switch 9 in your **ZSTU** startup routine to be certain **ZA** and **ZD** will be interpreted in Caché mode.

18.2.8 Extended Functions for DSM-11 Mode

Caché supports the following functions in both language modes:

- **$ZNext**
- **$ZOrder**
- **$ZPrevious**
- **$ZSort**

18.2.9 Unsupported DSM-11 Functions

Caché does not support the following DSM-11 functions:

- **$ZCall**
- **$ZUCI**
CAUTION: Because the $ZUCI function can be abbreviated as $ZU, you must be sure to remove all references to it from DSM11 applications that you port to Caché, so that Caché does not treat them as calls to the Caché $ZU(n,...) function.

18.2.10 Extended Special Variables for DSM-11 Mode

Caché supports the following special variables in both language modes:

- $ZOrder
- $ZReference
- $ZVersion

Caché supports these special variables only in DSM-11 mode:

- $ZA
- $ZB

Both $ZA and $ZB return DSM-11 values.

18.2.11 Extended Commands for DSM-11 Compatibility Mode

Caché supports the following DSM-11 command in both language modes:

- ZTRAP

Caché supports the following commands in DSM11 compatibility mode:

- ZPrint
- ZWrite
- ZBreak

ZPrint is identical to the Caché mode Print command.

ZWrite is identical to the argumentless Write command.

In DSM-11 M, the ZBreak ON and ZBreak OFF commands control the processing of argumentless Break commands. The $ZU(68,5,n) function simulates this DSM-11 behavior and works in both language modes.

Issuing $ZU(68,5,0) disables the processing of argumentless Break commands for a Job (similar to ZBreak OFF). Issuing $ZU(68,5,1) enables the processing of argumentless Break...
commands (similar to ZBreak ON). Issuing $ZU(68,5) without the n argument returns the current state of the switch without changing it.

Use the Caché Configuration Manager to make ignoring argumentless Break commands the default behavior for all jobs.

18.2.12 Error Handling for DSM-11 Compatibility Mode

18.2.12.1 Error Messages

In DSM11 mode, the error messages returned by the $ZERROR special variable use the same spelling as their DSM11 counterparts.

18.2.12.2 BREAK 2 Error Handling

In DSM11 mode, Caché supports DSM11 version 2 error handling. DSM11 version 2 style error handling provides that whenever an M error occurs the entire frame stack is cleared and a GoTo is executed to the error handling routine.

To preserve this method of error handling, use the BREAK 2 command to enable DSM-11 version 2 error handling. To revert to normal DSM11 error handling, issue the BREAK -2 command. You can invoke these commands in DSM11 compatibility mode only.

If a routine that is executing in DSM11 compatibility mode calls a normal language mode routine, such as a system utility, the Caché standard error handling method applies throughout the execution of that routine.

18.2.13 $TEXT Comment Lines

Caché supports the double semicolon feature that lets you include specific comment lines in the compiled result. This feature is necessary when the comment line is referenced by $Text and the routine is to be distributed without source code. Because DSM11 has no equivalent feature, $Textz in DSM-11 mode requires that the source code always be available.

18.3 DSM Language Compatibility

Caché operates in DSM mode when working with a DSM routine that has been ported. You need not make extensive changes in order to run DSM routines that have been ported to
Caché. This section discusses how various commands and functions operate in DSM mode. DSM Language mode is $ZU(55,5).

### 18.3.1 Porting Routines from DSM to Caché

Caché operates in DSM mode when working with a DSM routine that has been ported. To port routines from DSM to Caché use the DSM %RS utility to save native DSM applications to tape or disk. Then, restore the applications on the Caché system using the Caché Studio editor or %RI utility.

The %RI utility and the Restore utility with DSM format (available in the Caché Explorer) recognizes native DSM %RS files and loads (compiles and saves) them as DSM mode routines.

Keep the following points in mind when working with DSM mode routines on a Caché system:

- DSM and Caché routines can coexist in namespaces.
- Caché and DSM routines can call one another as desired. Caché adapts to the saved language mode of the executing routine on a per routine basis.
- Caché provides only runtime support for the DSM language. It does not support the DSM development environment, nor does it support call to DSM library utilities or any of the VMS related external routines ($ZCALLs).
- When you load a routine with the ZLoad command, Caché sets the language mode to match that of the routine.
- If you change the language mode with the $ZU(55,n) switch so that it does not match that of the loaded routine, and then attempt to do a ZInsert, you will get a <LANGUAGE MISMATCH> error message.

**Note:** Currently %RI and ^%rde report <SYNTAX> errors on some DSM language features in a DSM mode routine. This indicates that the syntax is not valid Caché ObjectScript syntax. However, the code will interpret correctly if it is a currently recognized DSM language feature in DSM compatibility mode.

### 18.3.2 Programming in DSM Language Mode

In Caché, the Close, Open, Use, and Job commands feature position-based parameter values separated by a colon (:) character. For example:

```
Use terminal:{{[margin]:[protocols]:[terminators]}]:"mnespace"
```
By contrast, in DSM the **Close, Open, Use, ZUse, and Job** commands feature keyword syntax where keywords and not position give meaning to parameter values. For example:

```
Use terminal:[keylist]:"mnespace"
```

where `keylist` is one of the following:

```
keyword [=value](keyword [=value][,...])
```

In each, the intent is to communicate information about operations to the specified device. For example the programmer may want to convert **Read** command input characters to uppercase. In Caché native language mode, this is done with the following syntax:

```
Use Open:(:"+U")
```

and in DSM language mode the following accomplishes the same thing:

```
Use Open:CONVERT
```

Currently DSM language mode supports certain keywords for terminal type devices. The following keyword is recognized but treated as a no-op for the **Open** command:

```
BLOCKSIZE=n
```

The following keywords are recognized and functional when applied to sequential files (Windows platforms only) by the **Open** command:

- DELETE
- NEWVERSION
- READONLY
- RECORDSIZE

The **DISCONNECT** keyword is recognized and functional when applied to sequential files by the **Use** command.

The following keywords are recognized and functional when applied to sequential files by the **Close** command:

- DELETE
- NODELETE
- RENAME

For discussions on how to use the above keywords, see the DSM documentation.
The following exceptions apply to Caché in DSM language mode:

- When closing a newly created file, DSM will delete the file if nothing has been written to it. However, in Caché the `Close` command in DSM language mode does not delete the file.

- The `RECORDSIZE` keyword specified with the `Open` command for sequential files currently affects `Read` operations only.

- When the `RENAME` keyword is specified with the `Close` command and the file to be closed is marked for delete, DSM deletes the file, then generates a `%DSM-E-RENAMEERR` error. However, in Caché the `Close` command in DSM language mode renames the file and does not delete it.

- When the `RENAME` and `DELETE` keywords are specified with the `Close` command, DSM deletes the file then generates a `%DSM-E-RENAMEERR` error. However, in Caché the `Close` command in DSM language mode deletes the file.

The following keywords are recognized and functional for the `Use` command:

- `CENABLE`
- `CLEARSRCR`
- `CONVERT`
- `CURSOR`
- `DEVTYPE=s`
- `DOWNSCRO`
- `ECHO`
- `ERASELIN`
- `LINE`
- `NOCENABL`
- `NOCONVERT`
- `NOCURSOR`
- `NOECHO`
- `NOLINE`
- `NOREADPFLU`
- `NOTYPE`
- READPFLU
- TERMINATOR=s
- TYPE
- UPSCROLL
- WIDTH=n
- X=n
- Y=n

BREAK 0 must be active to allow CENABLE and NOCENABL to have their intended effect. BREAK 0 is the default in Application Mode. However, BREAK 1 is the default in Programmer mode and is implicitly reasserted at every Programmer Mode prompt. Therefore BREAK 0 must be re-typed at every Programmer Mode prompt to interactively test the CENABLE and NOCENABL keywords.

The CLEARSCR, DOWNSCRO, ERASELIN, UPSCROLL, X=n, and Y=n keywords cause VT100 escape sequences to be written to the specified device and will only work with devices that recognize the sequences. When the specified device is a console window, Caché automatically translates the escape sequences into window operations that perform the desired result.

The LINE and NOLINE keywords enable and disable Caché read line recall. Read line recall may initially be disabled by default. Use $ZUTIL(68,11,1) to enable read line recall by default for your process.

In DSM, the TYPE and NOTYPE keywords have no effect unless the /TYPEAHEAD command line qualifier is in effect for the DSM process.

In DSM for DEC OSF/1 AXP the same is true, but the NOREADPFLU keyword can be used to accomplish dynamically the same thing as the /TYPEAHEAD command qualifier while READPFLU negates the /TYPEAHEAD behavior.

Therefore NOREADPFLU and READPFLU are provided in DSM compatibility mode so that NOREADPFLU can be specified for the principal device at application start up to accomplish the same behavior as the DSM /TYPEAHEAD qualifier.

DSM compatibility mode recognizes the following keywords but treats them as no-ops for Use:
- ESCAPE (in effect by default)
- FLUSH
- NOPACK
DSM language mode also supports several keywords for the `Job` command. The following keywords are recognized and functional:

- **DATA**
- **INPUT**
- **OUTPUT**

The following keywords are recognized but treated as no-ops for `Job`:

- **DETACHED** (also a no-op on DSM systems)
- **ERROR**
- **NAME**
- **PRIORITY**
- **OPTIONS**

DSM compatibility mode currently handles all other ANSI standard M commands except the **BREAK** and **View** commands, which according to the standard take implementor-specific arguments.

### 18.3.3 Device Control Mnemonic Spaces and Device Control Mnemonics

DSM compatibility mode in Caché provides a migration path for most DSM applications that use mnemonic spaces and device control mnemonics.

For applications that can be migrated, the application code itself need not be changed. In many cases a routine name change is all that is required and in the other cases only routine entry point bridges are required.

Device control mnemonics are keywords that are used with the **WRITE** / format character and the **READ** / format character to perform device control. For example, in a X3.64 compliant mnemonic space, the command

\[ \text{Write} \ /	ext{CUP}(1,1) \]

performs cursor positioning. In DSM, user defined mnemonic space tables associate the keywords in the mnemonic space with routine entry points that are called at runtime to perform
the operation. DSM applications associate a mnemonic space table and the keywords defined by it with a device by specifying the mnemonic space name with an Open or Use command.

Caché does not use a table to map the device control mnemonics in a mnemonic space to M routine entry points. Rather it infers the entry point label name from the control mnemonic (the control mnemonic is converted to uppercase to form the label name) and it uses the current mnemonic space name specified for the device as the entry point routine name.

For example, in the following sequence Caché calls back to the M routine entry point CUP^%X364 as a result of the WRITE / command:

```
Set DevZero=0
Use DevZero::"^%X364" Write /CUP(1,1)
```

If your user-defined mnemonic space tables specify case conversion for control mnemonics, you can update your application to run in DSM compatibility mode in Caché.

Further, if all your control mnemonics already match all the label names of their associated Caché ObjectScript routine entry points, then all that is necessary is that all labels be present in a single routine. If any control mnemonic does not match the label name of its associated Caché ObjectScript routine entry point then an entry point “bridge” whose label name does match the control mnemonic can be added. The entry point bridge needs only to call the original target Caché ObjectScript routine entry point passing along any expected parameters.

Of course, you can produce the same effect by changing the label name of the original entry point. Whether bridge entry point labels or original entry point labels, all must be present in a single Caché ObjectScript routine.

In DSM compatibility mode, Caché determines the name of the Caché ObjectScript routine that contains control mnemonic call back labels in one of two ways. When a specified mnemonic space name begins with the “^” character Caché assumes the mnemonic space name is the Caché ObjectScript routine name; otherwise it adds a “^%Z” prefix to the mnemonic space name to form the Caché ObjectScript routine name. This latter case will be the rule unless you update your DSM application code (usually a less desirable option).

The following sequence is an example of the latter case:

```
Set DevZero=0
Use DevZero::"ZTERM" Write /CUP(1,1)
```

In this sequence Caché performs a callback to the Caché ObjectScript routine entry point CUP^%ZZTERM as a result of the Write / command.

The names of the Caché ObjectScript routines containing control mnemonic callback entry points can be changed to adhere to this convention.
18.3.4 Other DSM Language Features Implemented in Compatibility Mode

The following DSM Z commands are recognized and functional:

- **ZAllocate**
- **ZDeallocate**
- **ZInsert**
- **ZLoad**
- **ZPrint**
- **ZRemove**
- **ZSave**
- **ZTrap** (generates an error but DSM style error handling and error messages are not currently implemented)
- **ZWrite** (only the argumentless form is implemented)

The DSM **ZUse** command is recognized but treated as a no-op. Note that **ZUse** keywords are not currently implemented.

The following DSM Z functions are recognized and functional:

- **$ZDate**
- **$ZNext**
- **$ZOrder**
- **$ZPrevious**
- **$ZSearch** (works only on Caché for Windows)
- **$ZSort** (a subscriptless global $ZSORT argument is not currently implemented)

The following DSM I/O related Z special variables are recognized and functional for terminal devices:

- **$ZA**
- **$ZB**
- **$ZCONTROLC**
• $ZIO (works only on Caché for Windows)

The following other DSM Z special variables are recognized and functional:

• $ECODE
• $ESTACK
• $ETRAP
• $QUIT
• $STACK
• $ZDEVTYPE
• $ZERROR (contains error information but DSM-style error handling and error messages are not currently implemented)
• $ZJOB
• $ZNAME
• $ZREFERENCE
• $ZTRAP
• $ZVERSION

The following DSM library external functions ($ZCALLs) are recognized and functional.

Mathematical:
• %ARCCOS
• %ARCSIN
• %ARCCTAN
• %BOOLEAN
• %COS
• %EXP
• %LOG
• %LOG10
• %MAX
• %MIN
18.3.5 View Command and $View Function

Since the View command and $View function require knowledge of the contents of disk or memory structures, which vary from one implementation to another, Caché does not support the View command or the $View function in DSM compatibility mode.

18.3.6 Database Conversion

The utility %dsmcvt converts a DSM database into an Caché database. When run, %dsmcvt prompts you for the directory containing the DSM database to be converted. The database is converted to an Caché database in the current namespace.
18.4 DSM-J Language Compatibility

DSM-J language mode is set using the `$ZUTIL(55,6)` function.

Caché using the DSM-J language mode supports the following keywords when applied to sequential files by the **Use** command:

- `KAN[JDEVICE]`
- `NOKAN[JDEVICE]`
- `KCODE=conversion_specification`
- `KON[ESCAPE]=escape_sequence`
- `KOFF[ESCAPE]=escape_sequence`
- `KPITCH=pitch_value`
- `KDIR[INPUT]`
- `NOKDIR[INPUT]`
- `KIN=switch (switch= “ON” or “OFF”)`

For a complete discussion of the above-listed command keywords, see the DSM-J documentation.

18.5 DTM Language Compatibility

Caché operates in DTM mode when working with a DTM routine that has been ported. This section discusses which commands and functions operate in DTM mode and notes any variations in how they operate. DTM Language mode is `$ZU(55,2)`.

18.5.1 Programming in DTM Compatibility Mode

In general, when you are programming in DTM compatibility mode, ObjectScript operates as it would on a native Open M [DTM] system. Routine line structure follows DTM rules, so that your routines can use DTM-specific language elements.

However, there are several points you should especially keep in mind when using DTM mode.
18.5.1.1 Operators

You can use the DTM-specific operators in DTM compatibility mode. These operators are:

| <>   | The record-forming operator. The record-forming operator creates a string which contains the pieces in between the angle brackets. Example: If $ZPIECE is equal to "^" and you issue Set x = <a, b>, then x equals "a_"^"_b" |
| <.n > | The field operator. The field operator sets a string equal to the n th piece of another string. Example: If s equals "s12^ab", then s.2 equals "ab" |

18.5.1.2 Job Command

In Open M [DTM] compatibility mode, the Job command operates as does the Job command on a native Open M [DTM] system. This means there is a substantial difference in the Job command parameters.

In Cache, the process parameters for Job can be four positional values. These are:

(default:switch:principal-input:principal-output)

| default | The default namespace for the jobbed process. |
| switch | A numeric value specifying whether M should pass your current symbol table to the new process. |
| principal-input | The principal input device for the process. |
| principal-output | The principal output device for the process. |

Because the parameters are positional, you must specify them in the order shown. If you omit a parameter that precedes a parameter you are including, you must include a colon as a placeholder for it. See the Job command in the Cache ObjectScript Language Reference for more information.

In DTM, the process parameters can be up to 10 values. Each is in the form:

keyword=value

| keyword | The name of a specific process parameter. |
| value | The value you want to assign the process parameter. |

The keywords you can use are:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvmem</td>
<td>The amount of memory (in bytes) to allocate for the jobbed process' symbol table.</td>
</tr>
<tr>
<td>lvmin</td>
<td>The minimum amount of memory (in bytes) to allocate for the symbol table.</td>
</tr>
<tr>
<td>lvmax</td>
<td>The maximum amount of memory (in bytes) to allocate for the symbol table.</td>
</tr>
<tr>
<td>name</td>
<td>The name to assign to the job.</td>
</tr>
<tr>
<td>node</td>
<td>The name of the network node on which the job is to run.</td>
</tr>
<tr>
<td>nspace</td>
<td>The namespace in which to execute the job.</td>
</tr>
<tr>
<td>pdev</td>
<td>The principal device of the job.</td>
</tr>
<tr>
<td>priority</td>
<td>The priority of the job (0 through 9).</td>
</tr>
<tr>
<td>strstk</td>
<td>The size of the string stack in bytes.</td>
</tr>
<tr>
<td>sysstk</td>
<td>The size of the system stack in bytes.</td>
</tr>
</tbody>
</table>

Because of the keywords, the parameters are not positional. To specify multiple process parameters, use a colon-separated list in the form:

```
(keyword=value[:keyword=value]...)  
```

You do not have to specify colons for parameters you do not specify.

### 18.5.1.3 I/O Commands

In DTM compatibility mode, the I/O commands (Open, Use, and Close) use DTM argument syntax. In general, use the following guidelines when you are in Caché compatibility mode:

- You can specify the devices you open, use, and close as positive integer values only. The values must be those assigned to them in the DTM .DEV file. You cannot use a device mnemonic.

- The parameters you employ with **Open** and **Use** are device specific. You specify each parameter in the form `keyword=value`. To specify some parameters and accept the defaults for others, use a colon to occupy each default position. You can omit colons after the last specified parameter. Leading colons are required. Trailing colons for missing parameters are unnecessary. In this example, **Open** accepts the default device parameters for the device being opened for all but the fifth parameter:

```
Open DEV(:::::param5)
```

- You can use a *timeout* with I/O commands. You can specify timeouts as integer values, decimal values, or as expressions that evaluate to integer or decimal values.
• You cannot use control mnemonics with DTM devices.

• You cannot use device parameters with the Close command. DTM does not recognize Close device parameters.

**18.5.1.4 View and $View**

In DTM compatibility mode, View and $View operate as they do on a DTM system.

**18.5.1.5 DSM-11 Compatibility Mode**

When your process is in DTM compatibility mode, you can further set it into DTM’s DSM-11 compatibility mode. You then are able to load and run DSM-11 routines. Use the ZZCOMPAT command to switch between DSM-11 mode and Open M [DTM] compatibility mode. Use the $ZZCOMPAT special variable to determine the mode in which your process is set.

**18.5.1.6 Structured System Variables**

You cannot use structured system variables in Open M [DTM] compatibility mode.

**18.5.2 DTM Compatibility Mode Commands, Functions, and Special Variables**

The following sections list the commands, functions, and special variables available in DTM compatibility mode.

**18.5.2.1 Commands**

These ANSI Standard commands operate identically in Caché mode and DTM compatibility mode:

• Do
• Else
• For
• Halt
• If
• Kill
• Lock
These ANSI Standard commands are not available in DTM compatibility mode:

- TCommit
- TRollback
- TStart

These extended commands operate identically in Caché mode and DTM compatibility mode:

- ZInsert
- ZNSpace
- ZRemove
- ZSync
- ZZDump

The following extended command is not available in Open M [DTM] compatibility mode:

- ZTrap

### 18.5.2.2 Functions

The following ANSI Standard functions operate identically in DTM compatibility and Caché mode:

- $ASCII
- $Char
- $Data
- $Find
- $FNumber
Open M Language Compatibility

- $Get
- $Justify
- $Length
- $Name
- $Next
- $Order
- $Piece
- $QLength
- $QSubscript
- $Query
- $Random
- $Reverse
- $Select
- $Translate
- $ZBitAnd
- $ZBitCount
- $ZBitFind
- $ZBitLen
- $ZBitNot
- $ZBitOr
- $ZBitSet
- $ZBitStr
- $ZBitXOr

These extended functions are not available in DTM compatibility mode:

- $ZBoolean
- $ZF
- $ZHex
• $ZIncr
• $ZNext
• $ZSearch
• $ZSort
• $ZTime
• $ZU

The following DTM mode math functions operate identically in Caché mode and DTM compatibility mode:

• $ZAbs
• $ZArcCos
• $ZArcSin
• $ZArcTan
• $ZCos
• $ZCot
• $ZCSC
• $ZExp
• $ZLn
• $ZLog
• $ZPower
• $ZSec
• $ZSin
• $ZSqr
• $ZTan

These set up a general error handler and call %math utility entry points. The general error handler turns any reported error into an <ILLEGAL VALUE> error which is returned to the calling routine.
18.5.2.3 Special Variables

The following ANSI Standard special variables are not available in DTM compatibility mode:

- $ECODE
- $ESTACK
- $ETRAP
- $QUIT
- $STACK

The following extended special variables operate identically in Caché mode and DTM compatibility mode:

- $ZHOROLOG
- $ZNSPACE

18.5.3 Database Conversion

The utility %dtmcvt converts a DTM database into an Caché database. When run, %dtmcvt prompts you for the directory containing the DTM database to be converted. The database is converted to an Caché database in the current namespace.

18.6 MSM Language Compatibility

Caché operates in MSM mode when working with a MSM routine that has been ported. This section discusses which commands and functions operate in MSM mode and notes any variations in how they operate. MSM Language mode is $ZU(55,8).

A language mode can be set individually for each routine and a routine compiled in one language mode can call or be called by a routine compiled in another mode. Thus, for instance, an MSM mode routine could call a DTM mode routine which could in turn call another MSM mode routine.

After an MSM application is compiled in the correct language mode, it can be installed and run on any Caché system, no matter what other applications or language modes are used on that system.
Almost all language mode processing occurs at compile time, not run time. As a result, using a language mode such as MSM will generally deliver the same high performance as Caché’s native language mode.

### 18.6.1 MSM Compatibility Mode Commands, Functions, and Special Variables

The following sections list the commands, functions, and special variables available in MSM compatibility mode.

#### 18.6.1.1 Commands

These ANSI Standard commands operate identically in Caché mode and MSM compatibility mode:

- Close
- Job
- Open
- Use

These extended commands operate identically in Caché mode and MSM compatibility mode:

- ZAllocate
- ZBreak
- ZDeallocate
- ZPrint
- ZTrap
- ZWrite

#### 18.6.1.2 Functions

These extended functions operate identically in MSM compatibility mode and Caché mode:

- $ZNext
- $ZOrder
- $ZPrevious
18.6.1.3 Special Variables

The following extended special variables operate identically in Caché mode and MSM compatibility mode:

- $ZA
- $ZB
- $ZC
- $ZERROR (with MSM error messages)
- $ZORDER
- $ZREFERENCE
- $ZVERSION

18.6.2 Database Conversion

The utility %MSMCVT converts an MSM database into an Caché database. When run, %MSMCVT prompts you for the directory containing the MSM database to be converted. The database is converted to an Caché database in the current namespace.